Requirements Engineering for Software Reuse

Kevin L. Mills

November 23, 1992

I. Introduction

Reuse of software components emerged as an industry goal subsequent to a NATO conference in 1969 where Doug McIlroy first introduced the concept. Over the two decades since, reuse remained a topic of much discussion and some research; and, although different views exist on the degree of success enjoyed by software developers in today's industry, most students of the state of software development practice agree that McIlroy's original vision has yet to be achieved and that increased reuse of software components is possible and remains a goal worth pursuing.

The present paper attempts to advance the cause of software reuse by investigating two main ideas. First, how can advances in requirements engineering be used to improve software reuse? This paper proposes that a requirements engineering process, and supporting techniques and tools, can be used to generate the domain knowledge that is a necessary condition for successful reuse. Further, this paper proposes that prototyping can be used during requirements engineering to evaluate the reusability of knowledge and components identified in earlier phases of the requirements engineering cycle. A secondary idea investigated in this paper is the degree to which reusable software can assist in prototyping during the requirements engineering process. For example, can we hope that reusable software

components will provide a basis for prototyping? These ideas are presented more fully in Section III, *Requirements Engineering for Reuse*.

Before considering the relationships between requirements engineering and reuse, some discussion is needed regarding reuse. Section II, *Reuse: Problems, Practice, and Potential*, presents the necessary material. First, a context is established by defining and limiting, for purposes of this paper, the scope of reuse. This includes an explanation of the motivations behind reuse and a brief evaluation of the progress achieved over twenty years. Second, problems that deter reuse are enumerated under four categories: 1) technical, 2) cognitive, 3) managerial, and 4) economic. A brief description of each problem is given. Third, some specific examples of reuse practice during the past decade are identified and described. The examples are taken from corporations, government organizations, and the mass, so-called "consumer", market. Fourth, some research aimed at overcoming problems associated with software reuse is reported. Section II then closes with some conclusions about reuse. The remainder of the paper investigates the theses advanced in Section III.

Section IV, *Domain Knowledge Acquisition and Representation,* considers how requirements engineering processes, and associated tools and techniques, for elicitation, organization, and representation of knowledge can support software reuse. Specifically, knowledge acquisition is explored as a method to elicit reusable concepts from domain experts, and knowledge representation is examined as a means to organize, describe, and refine domain knowledge. Section IV closes with some conclusions regarding knowledge acquisition and representation.

Section V, *Prototyping and Reuse*, explores possible links between prototyping and software reuse. The section begins with

an overview of prototyping approaches (by aim, by technical approach, and by life-cycle model), and then describes some specific prototyping systems reported in the literature within the last four years. The section closes with a discussion of potential relationships between prototyping and software reuse.

A concluding section (VI) provides a summary of the ideas advanced in the paper. Reuse is key to productivity improvements in most human endeavors. Software development is no exception. Reuse of software has improved over time, but greater potential for reuse appears feasible within the next decade or two. This paper propounds a view that a requirements engineering process, and related tools and techniques, can advance the state of software reuse.

II. Reuse: Problems, Practice, and Potential

Although software reuse is often considered to denote cobbling together a program from a set of software pieces or linking an application with a set of library subroutines, the reality of reuse defies simple description. As a working model of software reuse, Prieto-Diaz defines two levels: 1) ideas and knowledge and 2) artifacts and components. [PRIE87a] This is a convenient dichotomy because whenever a programmer creates software he is reusing knowledge he already possesses. [CURT89] On a larger scale, programming projects reuse a massive amount of knowledge, including software development process knowledge. Thus, initiatives such as that of the Software Engineering Institute to document, refine, and promote improved software development processes are an example of reuse of ideas and knowledge to develop software. Probably the most productive reuse of knowledge to develop software obtains today from reuse of trained software development personnel. [MEYE87]

Other examples of knowledge reuse for software development abound. A huge commercial market exists for books describing data structures and algorithms, and for teaching about the nature and application of those algorithms and data structures. [STAN84] Another example of knowledge reuse is adoption of and adherence to technical standards and conventions. [RICE89] Going even further toward tangible knowledge, buying commercial software, including so-called 4GLs, can be viewed as reuse of knowledge and ideas. [BOEH87] Brooks describes a burgeoning mass market for software programs that are applicable to specific tasks, and he proposes to:

> equip the computer-naive intellectual workers ... with personal computers and good ... writing, drawing, file, and spreadsheet programs and then [to] turn them loose. The same strategy, carried out with generalized

> mathematical and statistical packages and some simple programming capabilities, will also work for ... laboratory scientists. [BROO87, p.16-17]

Introduction of commercial software products blurs the line between knowledge and artifacts. Since software artifacts and components embody ideas and knowledge, the reuse levels introduced by Prieto-Diaz perhaps have more to do with representation: knowledge and ideas being intangible until they are represented; once represented in human-readable form, they become artifacts, and when they reach a machine-executable form they can be considered software components.

The key point of this discussion is that one needs to reuse more than code. (In fact, it is difficult to define reusable components apart from a context; and a context can include the requirements, a specification, a system architecture, another program or software subsystem, and a test plan and test cases. [CALD91]) Lenz considers the key reusable component to be a specification that includes a functional overview, a programmer interface (syntax and informal semantics), formal semantics, any constraints or dependencies, a description of the rationale for and characteristics of the design, and an example usage. [LENZ87] From a single specification, Lenz envisions many potential implementations. This view is reinforced by others who stress that the design and architecture are more reusable than code over the long-term. [WIRF90,HORO84,JONE84] Still, the end goal is to produce an executable computer program that satisfies a given set of requirements; thus, reusable software components, such as subroutine libraries, Ada packages, program generators, code skeletons and templates, and reconfigurable software systems, remain a necessary, tangible aspect of reuse.

For purposes of the present paper reuse is circumscribed within a bounds of machine-processability; thus, reusable components must be both tangible and computer-processable. This

definition requires that problems associated with capturing and representing knowledge must be addressed. The definition also stipulates that knowledge and ideas represented solely in human-accessible form are outside the circle of reuse drawn within the present paper. So, for example, if a human-readable design is captured and represented, then such design must be accessible through a computer-searchable index, or such design must identify one or more implementations that can be incorporated in, or that embody, a computer program. This, then, is software reuse: finding, accessing, evaluating, and using or adapting one or more software components to satisfy a given set of requirements.

Successful reuse of software components leads to increased productivity among software developers, to improved quality in the delivered products, and to more cost-effective software maintenance. [CAVA89] Such improvements could prove valuable to organizations that depend on computer software. Boehm estimates that by 1995 a 20% improvement in software productivity will be worth $90 billion worldwide. [BOEH87] And there is ample evidence to suspect that reuse can become a normal part of software development practice. For example, a study of business software systems at Raytheon Missile Systems Division found that 60% of all designs and code were redundant and could be reused. [LANG84] Another study of California commercial banking and insurance applications found that approximately 75% of the software functions were common to more than one program, and concluded that less than 15% of the code written for such applications is unique, novel, or specific; the remaining 85% appeared to be generic. [JONE84] Further, when examining the immense leaps forward in computer graphics made between 1954 and 1984, Standish found reasons to hope for equal progress in the future in a number of applications. [STAN84] Matsumoto also

reported in 1984 that 50% of the lines of code delivered in products from the Toshiba software factory were reused. [MATS84]

In that same year, Kernighan described the UNIX operating system as a set of reusable programs that can be composed via a shell and pipes, coupled with a reusable set of operating system services that could be accessed by programs in a variety of source languages through linkable subroutine libraries. [KERN84] In addition to the UNIX tools, Horowitz and Munson identified a number of widely used subroutine libraries for mathematical functions and numerical analysis, and described compiler generators, simulation languages, and parameterized software systems as, then current, software reuse successes. [HORO84] Also in 1984, spreadsheets were declared software reuse successes. [JONE84] By 1985, some were claiming that a new industry was emerging to support the design, development, distribution, and maintenance of reusable "Software-ICs". [LEDB85]

By 1987, advocates such as Biggerstaff saw software reuse as a great promise unfulfilled. Biggerstaff's experience was that well under half of delivered systems could be composed of reused code. [BIGG87] Intermetrics, owners of a reusable software library, reported that 33% of delivered code consisted of reused Ada packages. [BURT87] Prieto-Diaz found that most reusable components were small in number of lines of code, were simple in structure, had excellent documentation, and were written in the same software language as the new software system under development. [PRIE87]

Two years later, NASA projects were achieving software reuse rates of only 32%. That same year Curtis reported that software development practitioners, depending on their operational definition of reuse, had been reusing software for years, had just started a promising reuse program, or saw a need to sponsor much more research on reuse. [CURT89]

Painfully slow progress was made on reusing software between 1984 and 1989. In 1984, studies predicted that 60-85% of business software could be composed from reusable components, and some enlightened companies were even achieving 50% code reuse. Yet, five years later, the best reported software reuse rates were at about 33%. By 1991, software reuse practice was seen to be at a stage of awakening, moving slowly toward a period of early use. [CALD91] Why has progress been so slow?

Successful reuse of software requires overcoming a long list of hard problems. Meyer discusses software reuse problems in terms of technical issues, economic incentives, and programmer reluctance. [MEYE87] In Meyer's opinion the main roadblocks to successful reuse are technical. Without evaluating Meyer's opinion, immediate evidence supporting his view can be drawn from the following discussion because the list of technical problems impeding software reuse is double that for any other category; however, the relative significance of each particular problem is difficult to evaluate. The present paper classifies reuse problems in four categories: 1) technical, 2) cognitive, 3) management, and 4) economic. The description begins with the technical problems.

The *reuse population problem* comprises the current dearth of reusable components. Obtaining qualified candidates for reuse is difficult, and adapting submitted code to a reusable form is expensive. [CAVA89] Software is not often designed for reuse, and even when software is so designed, writing reusable software is difficult. [RAMA86, MEYE87] Code can be too specialized and often includes too many representational details. [STAN84] For example, Biggerstaff points out that:

> [m]odules become less ... reusable the more specific they become because it is more ... difficult to find an exact match of detailed specifics. Modules subtlely encode ... specific information about a variety of things: operating system, run-time library,

> hardware equipment, ... data packaging, interface packaging, and so forth. [BIGG87, p. 43]

And yet, separating a reusable software component from a specific context is difficult. [CALD91]

Another reason for the paucity of reusable components is a lack of producers. Most software development is conducted on a project basis; but projects will never be an appropriate place to create reusable software. Projects are hindered by a deadline focus, lack wide domain knowledge, and lack a reuse perspective. [CALD91] Production of reusable components is also inhibited by lack of accepted frameworks or system architectures into which components can be integrated. [WIRF90] Another factor working against the supply for reusable software is lack of demand; "too few software developers value or appreciate a quality library...." [GRIS91, p. 264]

A large supply of reusable components would not be a panacea, additional issues would elevate in significance. One such issue is the *classification problem*. By what attributes should reusable components be described and classified to enable effective search and retrieval by potential users? Several approaches have been proposed [BUTR87, PRIE87, CAVA89] None of these schemes appears particularly effective, and none seem necessarily superior to the others. Defining an approach that enables discrimination between very similar components is a particularly difficult classification problem. [PRIE87]

Assuming that software components can be adequately classified, the *location and retrieval problem* must be addressed. How can potentially appropriate reusable candidates be located and retrieved? The search space could be immense. Some means of factoring out specificity is required, so that the search space can be narrowed. [BIGG87] If specificity is factored out, then a means of mapping between a specification and appropriate implementations is needed. [BIGG87] Helping a

programmer retrieve a group of possible reuse candidates is achievable, but allowing a programmer to find the closest match against his stated requirements is much harder. [RAMA86] No matter what approaches are used, reuse libraries must be organized for quick search and access. [CURT89]

With a candidate set of reusable components in hand, the *evaluation problem* must be solved. There are two facets to this problem: how close to the requirements does each candidate match and how easily reusable is each candidate? There are some proposals for solving these problems [BURT87, PRIE87]. Reuser experience is significant because programmers will need to determine if reuse will require more work in a given situation than would a new implementation. [RAMA86] None of the existing proposals appears particularly effective, although one of them [PRIE87] attempts to vary the evaluation depending on a reuse experience profile for the programmers conducting the evaluation.

The significance of reuse experience among programmers was investigated in a study by Woodfield. [WOOD87] In the study, 51 developers (25 from industry and 26 from a university) were given 21 software components and asked to determine if each component could be reused to satisfy a particular specification. The study resulted in four findings. First, programmers untrained in reuse could not evaluate the ability of a reuse candidate to satisfy implementation criteria. Second, programmers untrained in reuse are influenced by some issues that are unimportant and are not influenced by some issues that are important. Third, no groups of programmers could be identified as performing significantly better or worse in judging reusability. Finally, if a programmer judged that the work needed to reuse code was less than 70% of the effort required to build the code from scratch, then the component was chosen for reuse.

Having successfully selected a reusable component, programmers typically must overcome the *adaptation problem*. A programmer must understand a component in order to modify it. [CURT89] Depending on the match between the programmer's specification and the reuse component, the software might require conversion for a different operating system or programming language or hardware environment. The interfaces available to the component might not match the interfaces expected. [NOVA92]

When required to adapt reusable code, the tendency among programmers is to copy and modify. [CAVA89] To avoid copying, a number of problems must be solved. For example, who owns and is responsible for the reusable component? How are the components maintained and synchronized with the release of products that incorporate the components? [LENZ87] How can reusable code be kept available in a form that works on multiple computing platforms? [CAVA89]

Selby investigated reuse at the National Aeronautics and Space Administration (NASA), examining 25 software systems ranging in size from 3,000 to 112,000 lines of code, and found adaptation to be an important factor affecting reuse. [SELB89] He compared newly developed modules to two classes of reused modules: extensively revised and slightly revised. He found that modules reused without revision had: 1) fewer calls to other modules per line of code, 2) simpler interfaces, 3) less interaction with human users, and 4) higher ratios of comments to lines of code. Completely reused modules were generally smaller, required less development effort, required fewer versions during development, and had more assignment statements.

Selby's investigation introduces the *granularity problem* identified by Biggerstaff. [BIGG87] Smaller, simpler components tend to be reused more because the population is large and evaluation and adaptation are easy, though finding smaller

components can be hard and the payoff is usually low. Larger components tend to be reused less often because the population is low and evaluation and adaptation are hard, though finding such components is easy and the payoff can be high. In general, small reusable components are less desirable. [LENZ87, WIRF90]

Granularity of reusable components influences the *composition problem*. To be successful, reuse schemes must provide "...robust mechanisms to insure reliable and meaningful parts composition." [RICE89, p. 125] Two different approaches exist to solve the composition problem. One approach relies on standards for communication and data interchange. [JONE84] In this model, reusable components, which are assumed to be fairly large, are connected together via communication channels, and data is exchanged between the components in a standard format. The second approach relies on a standard architecture into which components can be linked using a range of different mechanisms. [WIRF90, JONE84] The UNIX model for composition is a hybrid of these approaches. [KERN84]

Two other technical issues merit discussion: the *documentation and representation problem* and the *requirements specification problem*. The documentation requirements for reusable components are at least as rigorous as for any other software, probably more rigorous. The documentation must facilitate the understanding required to aid in the evaluation and adaptation of components; for large reusable components this is critically important and also very difficult. Maintenance is 70%-90% of the software life-cycle, and understanding is 50%-90% of the maintenance problem. [STAN84] Documentation must include a specification, a design, a design rationale, constraints on reusing the component, and test cases. [CALD91] How should this information be represented?

The final technical issue addressed here is the *requirements specification problem*. If the reader is not yet

humbled by the scope and depth of technical barriers facing software reuse and thinks that these will in due course be overcome, then consider the trigger for software reuse: user requirements statements. Users may express their requirements in a form that can disguise cues that might otherwise trigger recognition of appropriate reuse. [CURT89] Meyer was right, the technical barriers to reuse are indeed high; but there exist other impediments as well.

The *programmer acceptance problem* is well-known. [MEYE87, CURT89, SAGE90] Experienced programmers tend to view their work as creative, and they interpret reuse as routine application of old technology. Programmers also possess a certain pride of authorship and believe that they can do the job better than others. Programmers tend to distrust software developed by those they do not know. Also, the work required to understand the code of others, is not normally viewed by programmers as interesting. Programmers tend to believe that they will not get credit for work that incorporates large amounts of reusable code.

The *novice programmer problem* is a special cognitive issue. [CURT89] The short-term memory of humans can handle about seven, plus or minus two, concepts at a time. To overcome this problem, programmers chunk complex concepts together under labels, and then the mind can process seven labels. The labels refer to information stored in hierarchical, semantic networks in a programmer’s long-term memory. Expert programmers are better at encoding new information and at mapping, comparing, and analyzing the information against the broad base of knowledge that they already possess. This means that novice programmers, who can benefit the most from reusable software, are not adept at identifying, analyzing, and evaluating candidates for reuse.

Another cognitive difficulty is the *force-fit problem*. [CURT89] Programmers will often try to force the application requirements to fit a structure or pattern for which they know a solution, even if the solution fails to satisfy some of the original specifications. A related issue is the *generalization problem. [MEYE87*, CURT89] Abstracting general concepts out of specific implementations to form a reusable concept is a difficult mental exercise. Such generalization is often required because solutions that a programmer knows for one application domain might not transfer easily to another.

Of course, managers also play a role in software reuse, and unfortunately, reuse often suffers from a *management commitment problem*. Building a library of reusable components takes time and costs money. Managers can seldom identify the potential for a good return on the required investment. Even when managers are inclined to establish a program, and to evaluate the results as time goes by, the *measurement problem* interferes. [CAVA89] What are the measures that will demonstrate increased productivity and improved quality? If measures can be defined, how will the required data be obtained? How can return on investment by accurately determined?

The *return-on-investment problem* is one of the economic issues impeding software reuse. If a company delivers software that is too general and too reusable, then management may fear that they will not get the usual follow-on business of maintenance and enhancements. [MEYE87] Individual programmers or small companies that could specialize in reusable components of a limited scope and size have no sure means of collecting for their efforts because their code can be easily copied and distributed across communications links. [COX92] This could be called the *intellectual property protection problem*. In addition, purveyors of small, reusable components face a

marketing problem. How can you sell a reusable stack, for example? [COX92]

Despite this host of difficult problems, software reuse is practiced to varying degrees in both industry and government. In the following paragraphs some instances of reuse practice between 1984 and 1992 are identified and described, beginning with reuse in the business systems domain at Raytheon Missile Systems Division, circa 1984.

Langergan and Grasso [LANG84] report on a reuse program at Raytheon that resulted in average code reuse rates of 60%. The approach to reuse taken at Raytheon began with analysis of existing application code. Langergan and Grasso identified six major functions in business applications, and, after analyzing those functions, discerned seven program logic structures. From this analysis they defined two types of reusable components: functional modules and program logic structures. A preliminary analysis of the COBOL programs at Raytheon uncovered 3200 functional modules that support fifty applications. Since the initial analysis revealed a great potential for reuse, the investigators were given support to analyze over 5000 programs. The programs were analyzed by programming supervisors and were classified against a set of criteria provided by Langergan and Grasso. This initial classification identified 1,089 edit programs, 1,099 update programs, 2,433 report programs, 247 extract programs, 245 conversion programs, and 161 data fix programs, for a total of 5,274. After generalization by Langergan and Grasso, the programs were reclassified as 1,581 edits, 1,260 updates, and 2,433 reports. For each module the average lines of code, by type, was: 1) edit, 626, 2) update 798, and 3) report, 507.

Out of the 5,274 programs initially analyzed, programming supervisors selected 50 for detailed study. The study revealed that 40-60% of the code was redundant and could be standardized.

The investigators then formed three prototype COBOL program logic structures and began to practice code reuse. Initial results found reuse rates between 15% and 85%. When the paper was published in 1984, over 5500 logic structures had been developed, and average reuse rates were 60%. These appear to be the best software reuse results ever reported in the literature.

At about the same time that Langergan and Grasso were investigating and implementing reuse in the United States, Matsumoto was also pursuing reuse at a Toshiba software factory in Japan. [MATS84] Matsumoto, approaching reuse in a more theoretical fashion than Langergan and Grasso, defined three levels of abstraction: specification, design, and code. Reuse at Toshiba was facilitated by providing traceability between these levels. A method of presentation, called Forms, was defined for each level. Form(1,Q*) denotes the specification, and includes the objects, relationships, control algorithms, input/output transformations, constraints, and givens associated with the problem. A Form(1,Q*) description remains under strict change control. A Form(3,Q*) presentation is a generic implementation of a solution. Form(1,Q*) descriptions are stored in a searchable library which provides links to appropriate Form(3,Q*) code segments in a computer-aided software engineering (CASE) system. When developing new software, designers are required to complete a representation called Form(1,P) which is used to search the repository for a match. Should an appropriate, matching Form(1,Q*) be located, the designer is required to follow the links to the associated Form(3,Q*) and then to generate code from the given skeleton. Using this approach, Toshiba achieved a reuse rate of 50% of the lines of code in delivered products.

A third approach to software reuse extant in 1984 was the UNIX operating system. [KERN84] The UNIX operating system

advocates a software development style where many small, general purpose programs are constructed and then linked together in novel ways through composition operations (shell and pipes). To facilitate the construction of the small programs, UNIX includes a number of software libraries that provide access to operating system services. Over the eight years since 1984, the UNIX model has proven surprisingly durable. Many of the computers delivered today to industry and government require support for the UNIX operating system; in fact, the U.S. Government has defined a Federal standard, POSIX, based on the UNIX system.

By 1987, IBM apparently decided that reusable software might have merit, and began to implement reusable building blocks. [LENZ87] The aim of such building blocks was to encapsulate functionality, to present well-defined interfaces, and to achieve zero-defect quality. Each building block is represented by a single, detailed specification that could point to potentially many implementations. (IBM has a rather large and varied product line.) Although IBM started out to build link libraries this approach was soon abandoned in favor of macro-based, code templates. The link library approach had a number of flaws: 1) procedure call overhead was too great, 2) parameters were too generic and had to be repeated, and 3) compilers could provide no support for correctness checking in the user's program. The results reported in 1987 were surprisingly modest. Only sixteen abstract data types, four procedural building blocks, and three functional building blocks had been created. Two implementations existed for some of the specifications. The component sizes ranged from 100 to 3000 lines of code, with 1000 being the average size. Two lessons were reported from this approach: 1) a clear understanding of the application domain is required to build reusable software and 2) fewer, larger components are easier to manage than a large number of small units.

Another reuse project reported in 1987 was conducted by Intermetrics. [BURT87] The Intermetrics reusable software library (RSL) relied mainly on Ada, although components in other languages were also accepted. Each reusable component was characterized according to 14 attributes, some of which were automatically validated by a supporting RSL database (RSLDB). Some of the attributes could only be verified by a human quality assurance expert. The attributes were screened carefully because the resulting information was used by a program called SCORE to assist users in retrieving components that might match specific requirements. Again, the reported results were somewhat disappointing when compared with the successes reported in 1984. Only 33% of delivered code consisted of reused Ada packages. Several of the delivered products showed poor performance until performance analyzers were used to profile the code, following which the code was tailored for the new application. Three lessons were reported by Burton: 1) standards are needed to define reuse attributes and metrics, 2) automated tools require a thorough understanding of the application domain in order to effectively evaluate the reuse potential of particular components, and 3) the value of a reusable library increases when integrated with supporting automated tools.

In 1989, Cavaliere reported on a software reuse project started at the Hartford Insurance Group in 1981. [CAVA89] The approach at the Hartford was similar to that of Langergan and Grasso. COBOL program skeletons, logic skeletons, and common functional modules were defined and created, although code generators were also used -- primarily for terminal screens and report formats. The innovations in the Hartford approach were mainly in areas of management. For example, a Reusable Code Review Board was established to accept, review, evaluate, and refine suggestions for reusable code, to identify and pilot new

reuse needs, and to conduct demonstrations and handle questions. One member of the board came from each application programming division at the Hartford. Each developer of new software was asked to submit information regarding the potential reusability of the new code, and management initiated an incentive program to give cash awards and recognition for reusable code submissions. Further, management invested in training programs and bulletin boards and established a reuse resource center that was manned 24 hours a day (programmers work at all hours) to help solve problems and answer questions. Cavaliere made the following recommendations to organizations considering a reuse program: 1) use 4GLs as much as possible, 2) develop and maintain an automated index of existing programs and functions, 3) provide full-time resources to start and support the program, 4) provide resources to assess changes in productivity and quality, and 5) setup a reusability users group.

By 1991, even the Department of Defense had established a software reuse repository. [DISA91] The Defense Information Systems Agency (DISA) provides government users and government contractors with dialup access to a database of reusable software components. The database can be searched with keys, assigned specific weights by the searcher, that guide access through an ten-facet classification scheme (component type, function, object, language, algorithm, data representation, unit type, certification level, environment, and originator). The repository can hold information on anything from functional specifications to code, and languages range from 2167 (a DOD specification standard) to COBOL. Apparently, each component is self-classified by the submitter, and some components are controlled by commercial licenses. The repository also maintains a set of metrics for each component meant to characterize reusability, maintainability, reliability, and

portability. Information is included describing known uses of each component, as well as known problems.

While the DOD appears to be a latecomer to software reuse, even such a successful computer and instrument company as Hewlett-Packard (HP) only initiated a Corporate Engineering Software Reuse program in October of 1990. [GRIS91] The program appears to have achieved little as of October 1991. Some HP divisions have started a multi-divisional domain analysis aimed at defining common architectures, components, and libraries for firmware in instruments and for chemical and medical applications. One goal is to produce frameworks and major components that can be reused across several product lines. A second goal is to develop a reuse education curriculum. Reuse at HP is envisioned to be independent of any specific programming language, but object-oriented analysis (OOA) and design (OOD) will probably be the basis for most of the domain analyses conducted in support of HP reuse objectives.

Looking beyond HP, a cursory review of trade magazines and the desks in homes and offices across the country reveals a large market for reusable software programs that run on particular computer architectures and under control of some specific operating systems. The mass market for reusable software provides applications such as spreadsheets, graphical user interfaces, communications programs, wordprocessors, databases, math programs, computer-aided design, and more. The market value of this software indicates an immense reuse success, a success built on de facto standards for hardware and operating systems. Techniques are now reaching the marketplace to enable these various applications to be composed in ways not originally intended by the application programmer. Some skeptics do not believe that a commercial mass market can achieve acceptable levels of reuse over the long-term. For example, Adele Goldberg believes:

> ...reusable, combinable applications, [t]oday we see this idea being promoted at the level of operating systems, window systems, and independent software architectures (low-level such as Microsoft's DLL and Sun's sharable libraries, and high-level such as Patriot Partners' Constellation project and ParcPlace's object model and frameworks approach). A public market is a very loosely organized environment...even well-designed components with minimally constrained interfaces will have trouble attracting a critical mass of customers. On the other hand, within a single organization, reusable components can be developed and redesigned with a context that can span a large fraction of their intended uses. In this way accumulation of reusable code can become an important business asset, and can be treated as an investment and a capital good, rather than simply a cost. [GRIS91, p. 268]

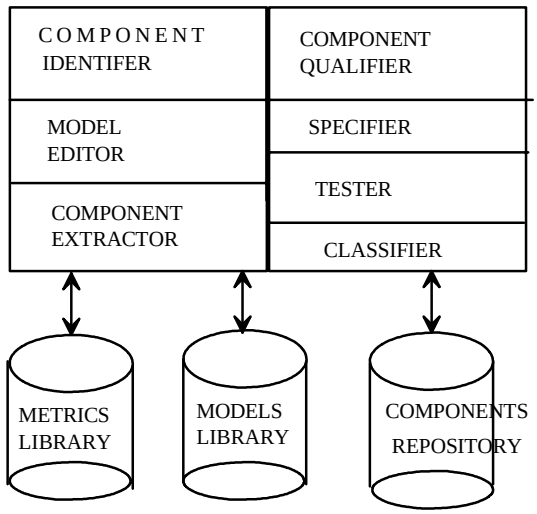
This tension between mass-market, reusable software and custom-developed, reusable software will continue for at least a decade. The buy or build decisions facing managers in large corporations and government organizations could have unforeseen effects on ideas about software reuse. In the meantime, software reuse, as originally envisioned by McIlroy, continues to advance in both industry and government.

Software reuse, as currently practiced, can be improved, and some of the problems described earlier may be overcome as the result of research aimed at establishing a foundation for software reuse. Two approaches to identifying reusable components are being explored: domain analysis and software re-manufacturing. Domain analysis is a front-end activity analogous to elicitation, organization, and analysis in the requirements engineering process. (See, for example, the work of Prieto-Diaz. [PRIE87a]) Since domain analysis forms an integral part of the proposals advanced later in this paper, a discussion of the topic will be deferred until then. Software

re-manufacturing, however, falls beyond the scope of the proposals made in the present paper, and is, therefore, considered here in the presentation of background material relating to reuse.

Caldiera and Basili have advanced specific proposals for re-manufacturing reusable software from existing code. [CALD91] This idea appears attractive because some of the most successful reuse projects reported in the literature started with existing COBOL code. The novel aspects of the proposals from Caldiera and Basili are three: 1) reuse is addressed independent of a specific domain, 2) software metrics are used to help identify reusable components, and 3) an automated system, CARE, extracts reusable components from existing code. Their current implementation supports analysis of ANSI C and Ada, and they have built a component extractor for C programs. A description of the system is given in Figure II-1 below.

Figure II-1. CARE, a reusable software components extraction system. [CALD91, p.76]



Caldiera and Basili intend that their tools support a reusable software factory operating in a life-cycle independent from that of specific projects. In their model, software projects build applications and reuse components, while the factory handles requests for reusable parts and builds, or

extracts, and packages reusable components. Although their extraction tools operate without specific domain knowledge, Caldiera and Basili point out that workers in a reusable software factory must have intimate knowledge of the domain.

Notice in Figure II-1 the tool called a classifier. This part of CARE, intended to assist in the problem of classification of reusable components, is not yet designed. In fact, classification of components is another area of research surrounding reuse. Prieto-Diaz and Freeman have proposed "...a faceted classification scheme, based on reusability-related attributes, and a selection mechanism." [PRIE87] The goal of their research was to aid programmers to distinguish between very similar reusable components. Their approach integrates a classification scheme with automated tools to help search for reuse candidates and to help evaluate the modification effort required to reuse candidate components. They criticize the existing, enumerative schemes (circa 1987) as vague, or as excluding consideration of reuse attributes, focusing instead on application and hardware type. They also demonstrate that faceted classification schemes are more easily expanded than the enumerative approaches (e.g., the Dewey Decimal System). Their approach appears to be implemented by the DOD software repository described earlier, except that their ideas about including a reuser experience profile have not been implemented, probably because the DOD repository is meant to serve a large, almost public, audience. Despite the best efforts of Prieto-Diaz, Freeman, and others, the classification problem remains a research issue, and the problem continues to limit the potential of software reuse.

Another area of research involves techniques for constructing programs from reusable knowledge or from reusable components. Approaches to generation of programs from reusable knowledge are either: 1) automatic generation from a

user-written specification [e.g., BARS85] or 2) automated assistance for a human programmer [e.g., RICH88]. The automatic generation approach appears to require substantial knowledge of a fairly narrow domain. The automated assistant approach appears to depend on deep knowledge at many levels, as well as some integration between the levels of knowledge. In general, automatic programming systems do not handle well certain aspects of software requirements that are important and visible to the user, for example, the details of the user interface or the myriad, specific exception conditions that can occur. Another facet of software development that automated programming systems must address is the iterative nature of the process. Development usually moves from vague concepts toward a software solution in iterative cycles that get closer and closer to an acceptable solution. [RICH88a] "Automatic programming systems will have to explain what they have done and why." [RICH88a, p.43] Early automatic programming systems have not handled this interaction and iteration particularly well. One reason for failing in the area of user interaction is that automatic programming systems traditionally depend upon specific languages that are not easy for users to understand.

An alternative to automatic programming is composition of a software system from reusable components. In the past, this approach was limited by the problems inherent in link libraries: too many components that are too small, incompatible interfaces among components, inability to detect errors at compile time, and limited functionality among available libraries. Research into object oriented techniques may overcome some of the limitations of link libraries. [WIRF90] A recent study found that use of an object oriented paradigm improves software development productivity, and that a significant part of the improvement was due to reuse. [LEWI91] The object oriented model encapsulates groups of functions and attributes into a

software unit, typically called a class, that is meant to be reused to implement a concept. This approach is seen as superior to the link library approach because a class provides a logical grouping of functions that can help programmers better understand the relationships between functions and attributes. Interestingly, the latest research on object oriented techniques describes the class as being too fine-grained. [WIRF90] Instead, researchers are working to define frameworks, or ensembles, that group classes into bigger units and then concentrate on defining the interfaces between the groupings. The seed for this idea probably grew out of the development of graphic user interfaces (GUIs) which were implemented as a related set of object classes.

An example of object oriented frameworks outside the domain of GUIs is Choices, an operating system framework written in C++. Choices includes an interlocking set of frameworks for file systems, virtual memory, communications, and process scheduling. The frameworks can be instantiated as a Berkeley UNIX, System V UNIX, or MSDOS system. Experience with Choices provides some interesting insight into the potential for and limitations of object oriented approaches to software reuse.

> The Choices file system has gone through many versions, and each version is more general and reusable than the previous one. The core classes have been stable for some time, while the outer classes are newer and still changing. This is typical of reusable designs. Reusing the early versions points out design weaknesses that must be corrected. A framework's designer can be confident of its reusability only after it has been successfully reused several times. Designing a framework is itself research. The designer must understand the possible design decisions and must organize them in a set of classes related by the client/server, whole/part, subclass/superclass relationships. Thus, the designer is developing a theory of the problem domain

> and expressing it with object oriented design. [WIRF90, p. 118]

The early research on object frameworks has identified the need for automated tools that can configure applications from a framework and that can compose applications from a set of configured frameworks. Some approaches being investigated include scripting languages and visual scripting tools. Adequate support for composition tools will probably require knowledge-based systems such as those envisioned by Rich and Waters. [RICH88a]

Rich and Waters advocate a hybrid approach to software development from reusable components. They envision an automated assistant that will help the human programmer (user or professional) synthesize a software solution by inspecting various chunks of knowledge (called clichés) and by selecting a set that can be integrated to satisfy a given set of requirements. Their approach requires that an automated assistant possess knowledge of application domain clichés and programming clichés. In effect, domain knowledge and programming knowledge would be used by an expert system to help a programmer assemble a solution from reusable parts.

What, then, can be concluded about software reuse? Reuse success stories possess some common traits: 1) the application domain is well-understood, 2) components tend to be large, and 3) a definite system model or architecture exists into which components can be fitted. Knowledge of both the application domain and of programming technology is necessary for successful software reuse. As a corollary, knowledge, while necessary for software development through reuse, is not a sufficient condition; the knowledge must be represented in a form from which programs can be generated or composed. This indicates that knowledge-based systems (KBS) will play an essential role in successful software reuse in the future. KBS can perhaps

assist with the *evaluation and adaptation problems* by aiding the programmer to match requirements to available software solutions, or components, and by helping adapt existing solutions and components to specific requirements. KBS should also ameliorate the *force-fit and novice programmer problems* by interacting in an iterative cycle with the programmer to identify the best solutions for particular requirements.

Although the *classification problem* seems unsolvable, viewing resuable components in larger chunks promises to change the nature of the problem. When dealing in larger components, classification seems more likely to focus on significant concepts in the problem domain. Current classification schemes focus too much on solution spaces (e.g., environment constraints, representation form, data structues, and algorithms). Creating fewer, larger, reusable components should also limit the effects of the *location and retrieval problems*.

Finally, the requirement for a specific system architecture or framework to support software reuse suggests that prototyping systems and standard hardware and software platforms can play a significant role in developing systems from reusable software. Prototyping systems might establish an architecture into which reusable components can be fitted and evaluated.

Over the more than two decades since McIlroy first introduced the prospect of a reusable software components industry, significant trends in lower-cost, higher-performance hardware have increased demand for software. The software delivered today embodies richer functionality than was feasible in 1969. Even with these changes, the apex of software reuse seems to have occurred in 1984, when at least two practitioners reported a 50% reuse rate in two different application domains. Advances in software languages and technology seemed to wash away these gains so that by 1987 the reuse rate among typical software projects had dropped to 33%. Notably, reuse in the

1987 to 1989 period appears to have relied on smaller, simpler functional modules, the concept of reusable program logic and control structures seems to have vanished. The newest efforts beginning in 1990 appear destined to depend on object oriented programming as the key to reusable software. Yet, object oriented approaches are already being called into questions by some researchers because classes, even though they can encapsulate many functions and attributes, define reusable components that are too small. Although progress on software reuse appears to be moving in reverse, sometimes backward movement precedes substantial progress.

This paper argues that the problems plaguing software reuse are well known, that software reuse during the mid-1980's revealed some essential requirements for success, and that emerging processes for requirements engineering, and associated tools and techniques, can be applied to boost software productivity and quality by facilitating increases in reuse of software.

III. Requirements Engineering for Reuse

As experience was gained with software reuse, researchers came to understand that intimate knowledge of an application domain is a necessary prerequisite to identifying, defining, and specifying reusable software components. Out of this understanding is emerging a new discipline dubbed domain modeling and analysis. This paper argues that domain modeling and analysis is nothing more than a requirements engineering process with a slight difference in emphases. Further, this paper proposes that a requirements engineering process, and related tools and techniques, can support the modeling and analysis of application domains. In fact, this paper identifies requirements engineering steps, tools, and techniques that can improve the effectiveness of domain modeling and analysis to a greater degree than any processes, tools, or techniques developed to date specifically for the analysis of application domains. The material presented in this paper suggests that a domain modeling and analysis process should be merged into the requirements engineering process, strengthening both processes and permitting increased benefits from automated tools that can support both processes.

These arguments are presented in the following fashion. First, a brief summary of the requirements engineering process is provided. Second, a discussion of domain modeling and analysis is presented. Third, a mapping between the domain analysis process and the requirements engineering process is proposed. Finally, requirements engineering tools and techniques that can be applied to analyze and model application domains are identified.

The requirements engineering process begins with a concept or idea, however vague, that might benefit from an automated

solution. The process ends when a detailed, written specification of software requirements (called the SRS) is produced. The process operating between these two mileposts comprises an iterative set of steps. Various proposals exist for naming and defining these steps, but a great deal of agreement exists about the activities that must be completed to move through the process. [DAVI90, FREE80, RAMA86, SAGE90] This paper assumes a requirements engineering process model defined for a George Mason University Ph.D. seminar, "Topics in Requirements Engineering," led by Dr. James Palmer and held during the fall semester of 1992. Figure III-1 illustrates the process model.

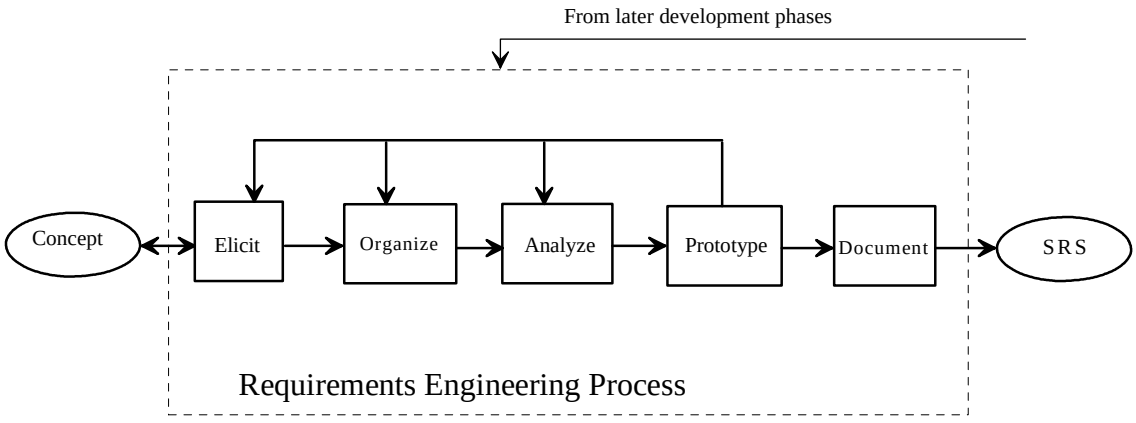


Figure III-1. Requirements Engineering Process Model

The first step in the requirements engineering process involves elicitation of requirements from those who can refine the concept. This step is both iterative and interactive. Usually, one or more analysts skilled in the problem domain are assigned to meet with users and managers, to review documentation, to experiment with any existing automated or manual system. Elicitation can recur often during the requirements engineering process; note in Figure III-1 that elicitation can be reactivated from several other steps in the process. The main aim of elicitation is to accumulate as much information about the requirements as possible, ensuring that

the information is expressed in terms of the problem domain. A secondary aim of elicitation is to build trust and rapport between the elicitors and the users, so that future iterations of the elicitation process will be productive.

The second step in the requirements engineering process encompasses organization of the material gathered during elicitation. The raw requirements from the elicitation step likely contain redundancies, inconsistencies, omissions, and ambiguities. The organization step is the first opportunity to discover some of these problems. (The earlier in the process that errors are uncovered, the easier and cheaper they are to fix.) Organization of the requirements also provides the beginning of understanding the problem by imposing a structure on the raw requirements. The aim of the organization step is uncover errors in the raw requirements and to begin the process of model building that continues during the analysis phase.

The third step in the requirements engineering process, analysis, comprises constructing a model of the problem in a form that can be exercised, exercising the model, and discovering requirements errors: incompleteness, ambiguity, conflict, redundancy, and imprecision. These discoveries are used to return to the elicitation step with a set of specific issues that can be discussed with the users. Also during the analysis phase, problems in organization of the requirements can be uncovered, initiating a return to the organization step to refine the model. When the analysis is supported by automated tools, the requirements must be transformed from natural language to a more formal notation. The process of making this transformation is itself a form of analysis that can help discover errors in the requirements; however, care must be exercised because analysts have a natural tendency to interpret the requirements during this transformation. A more prudent

course is to iterate through the elicitation cycle, getting the user's view, rather than making a hidden interpretation.

The fourth step, most often seen as optional, employs some form of prototyping to evaluate issues pending from the analysis. For example, the requirements might be a bit unusual; thus, a proof-of-concept could be prudent. More typically, the particular user interface would be implemented and exercised, in effect prototyping to elicit user views on the interface, or to evaluate the effectiveness of the interface. Some requirements might reflect concerns about performance or some delicate algorithms: these could be explored through prototypes. A growing, but still minority, position views prototyping as iterative development of the actual application. This view is usually held only where the problem domain is fairly routine, and where performance is not a critical issue.

The final step, documenting, achieves a large transformation of the information obtained during the preceding steps. The requirements are transformed into a Software Requirements Specification (SRS) intended to guide the design phase of software development. The transformation made during this step moves the requirements from the problem domain into a solution domain. Completion of the SRS does not end the requirements engineering process; these steps may be revisited during later stages of the development.

The requirements engineering process is focused and specific. The aim is to elicit, organize, and analyze requirements for a particular software system, and to transform those requirements into a form, the SRS, that can guide software development. How does this compare with processes for domain analysis?

Domain analysis attempts to generalize all systems in an application domain, that is, to produce a *domain model* that transcends specific applications. [PRIE87a] Prieto-Diaz

envisions domain analysis as a process that precedes requirements engineering for specific systems. In his view, the domain model that results from the analysis can help elicit, organize, and analyze the requirements for a specific system in an application domain. This view is shared by Iscoe. [ISCO88] Iscoe points out that "...lack of a formal model to represent information at the application domain level results in a severe information loss during the mapping process..." that creates the SRS. [ISCO88, p. 300] Iscoe goes on to demonstrate how domain knowledge can be used to detect ambiguities and omissions.

Although no accepted definition of the form of a domain model exists, remarkable similarity can be seen among researchers regarding the content of a domain model. Jacobson and Lindstrom describe a domain model as the set of domain objects (including their attributes and functions) and the relationships between them. [JACO91] This description mirrors that of other domain analysis advocates. [ARAN89, ISCO88, PRIE87a] Iscoe adds to his description of a domain model the set of rules that can be used to compose, generalize, and specialize domain objects. [ISCO88]

The disagreement among researchers regarding a representational form for a domain model seems to be motivated by differences in the use that each intends for the model. For example, Iscoe hopes to generate inputs into any of a number of transformational program generators; therefore, he envisions a model that captures the relevant domain knowledge needed for domain-specific application programming. (Not that he knows what domain knowledge is relevant.) So, Iscoe’s research aims to define a model for representing domain knowledge. (He omits completely consideration of programming in the large, in favor of small manageable application domains.)

Alternatively, Prieto-Diaz aims to create specific, unique languages for each domain that is modeled. The language becomes

the domain model and is used to describe objects and operations common to the domain. As Prieto-Diaz explains: "if a domain language exists that can acceptably describe the objects and operations of a required system, then the systems analyst has a framework on which to hang the new specification." [PRIE87a, p. 23]

Jacobson and Lindstrom prefer a graph representation of the domain because they aim to build a model that facilitates reasoning about system modifications. [JACO91] Arango is motivated by software reuse, and thus his model of the domain includes software reusability information. [ARAN89]

Prieto-Diaz places the issue of domain model representation in perspective:

> Selection of a particular representation structure would depend on the kind of domain analyzed. Different forms could be used within the same domain depending on its size. A high level domain model could be in the form of a faceted scheme or a simple hierarchy with semantic networks and frames used for lower level domain elements. [PRIE87a, p. 28]

This issue of representation of the output of a domain analysis process is rather revealing. Since an SRS embodies a natural language document that is meant to guide further development by human, software designers, a vary of representations can be used effectively. As with domain models, wide agreement exists on the general content of an SRS. The main differences in SRS documents are variations in form. With respect to domain models, however, issues of representation appear paramount. Domain models must be both understandable by humans and processable by a computer. In addition, models of a particular domain usually must be integrated with models from other domains. Such integration can be impeded by incompatible representations. Because domain models are sensitive to issues

of representation, the domain modeling activity must precede creation of a SRS; therefore, domain analysis must either precede, or be contained within, the requirements engineering process. Before considering the relationship between the domain analysis and requirements engineering processes, a discussion of the domain analysis process is in order.

A number of researchers have proposed processes for domain analysis. McCain proposes a four step process: 1) define reusable entities, 2) create reusable abstractions from these entities, 3) perform a classification of the abstractions, and 4) define an abstract interface, the essential constraints and limitations, and any customization requirements for the abstractions. [PRIE87a] Arango proposes a six step process: 1) bound the domain, 2) collect standard examples of implementations from the domain, 3) perform a systems analysis on each example, 4) identify potential abstractions, 5) map the abstractions into a formal representation using conceptual modeling languages, and 6) identify potential abstractions that can lead to multiple implementations. [ARAN89] Probably the most complete domain analysis process model has been proposed by Prieto-Diaz. [PRIE87a]

Prieto-Diaz divides the domain analysis process into three phases: pre-analysis, analysis and post-analysis. The purpose of the pre-analysis phase is to define and scope the domain, to identify sources of knowledge and information about the domain, and to define a strategy for the analysis. The analysis consists in finding abstractions for groups or classes of groups, documenting these abstractions as frames, classifying and seeking relationships between the frames, and documenting the relationships as a taxonomy of the domain and as a structure of relationships that can be used as a domain model. The post-analysis phase covers encapsulating the reusable abstractions and producing guidelines for reusing the

abstractions. Prieto-Diaz illustrates his proposed process with a set of data flow diagrams that help to understand the inputs to the process, the specific transformations or activities, and the results. These diagrams are repeated here as Figures III-2, III-3, III-4, III-5, III-6.

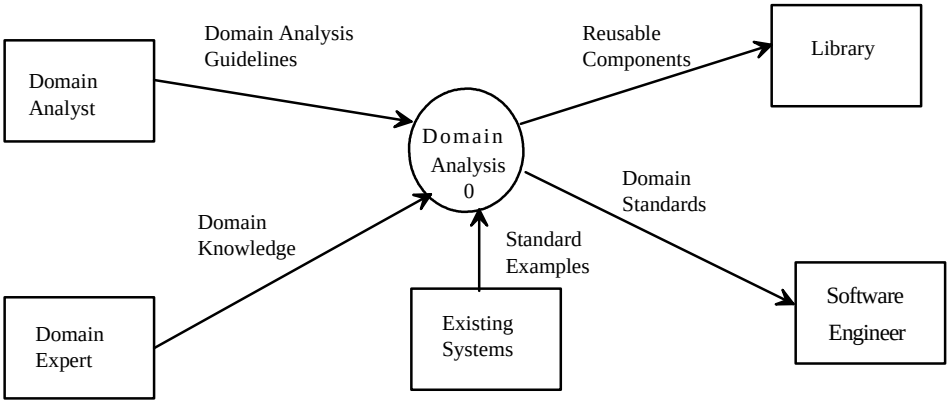


Figure III-2. Domain Analysis Context Diagram [PRIE87a, p27]

Figure III-2 provides a context diagram for the domain analysis process proposed by Prieto-Diaz. Note that the process involves a domain expert and a domain analyst on the front-end, and that a set of reusable components is output, as well as guidelines for software engineers to use the reusable components. In addition to domain knowledge, input comes in the form of examples from existing systems. Figure III-3 decomposes

the context diagram.

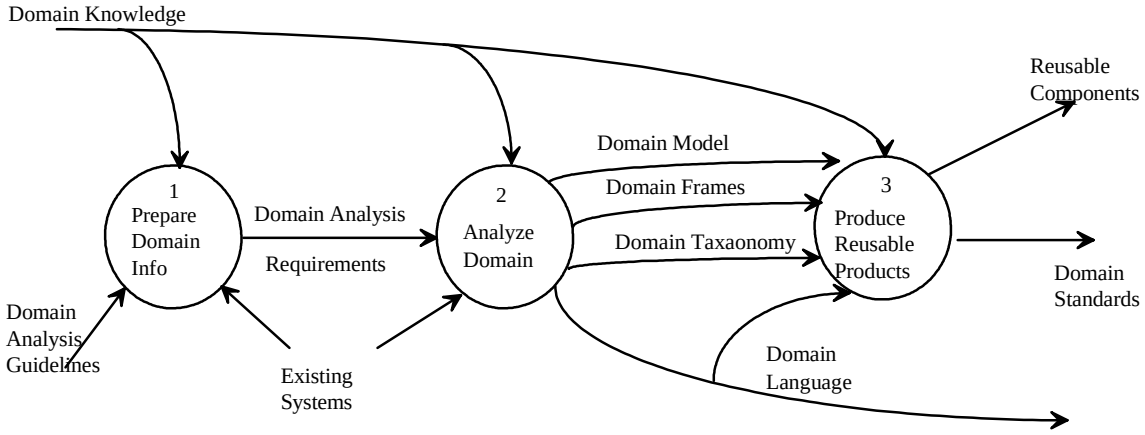


Figure III-3. Domain Analysis Level 0 [PRIE87a, p. 27]

Figure III-3 corresponds to the pre-analysis, analysis, and post analysis phases proposed by Prieto-Diaz. Each of these phases is decomposed further in Figures III-4, III-5, and III-6.

When domain analysis is conceived as a preliminary activity that precedes requirements analysis, some interesting issues arise. For example, who conducts the analysis? Project teams are usually focused on producing a specific product related to the goals and objectives of the project. Forming special domain analysis teams raises other questions: Who pays for the analysis? What are the specific concrete outputs of the analysis? Who evaluates these outputs? How? These issues are very similar to issues raised in an earlier discussion about remanufacturing reusable software components (see pages 21 to 22).

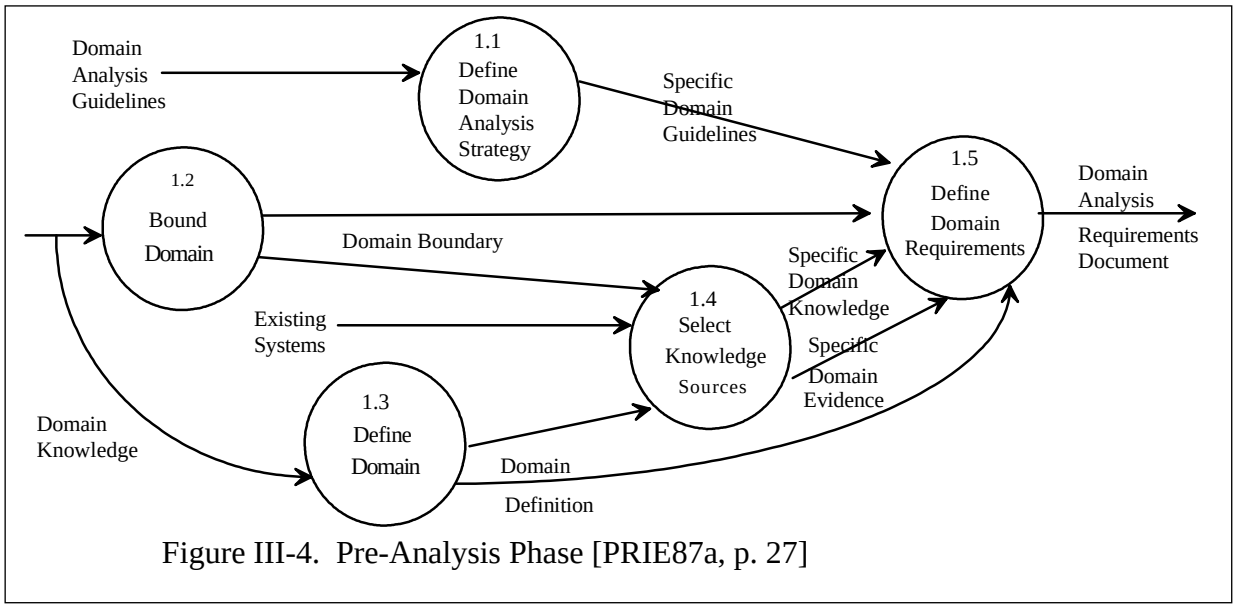


Figure III-4. Pre-Analysis Phase [PRIE87a, p. 27]

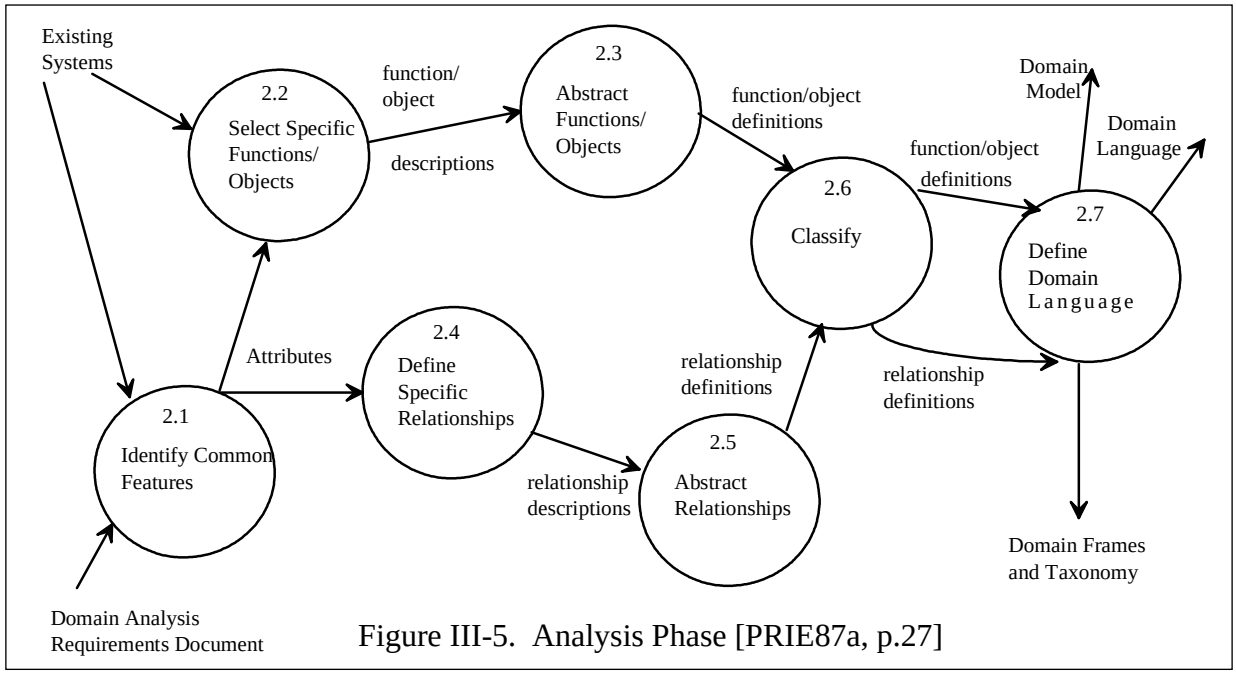


Figure III-5. Analysis Phase [PRIE87a, p.27]

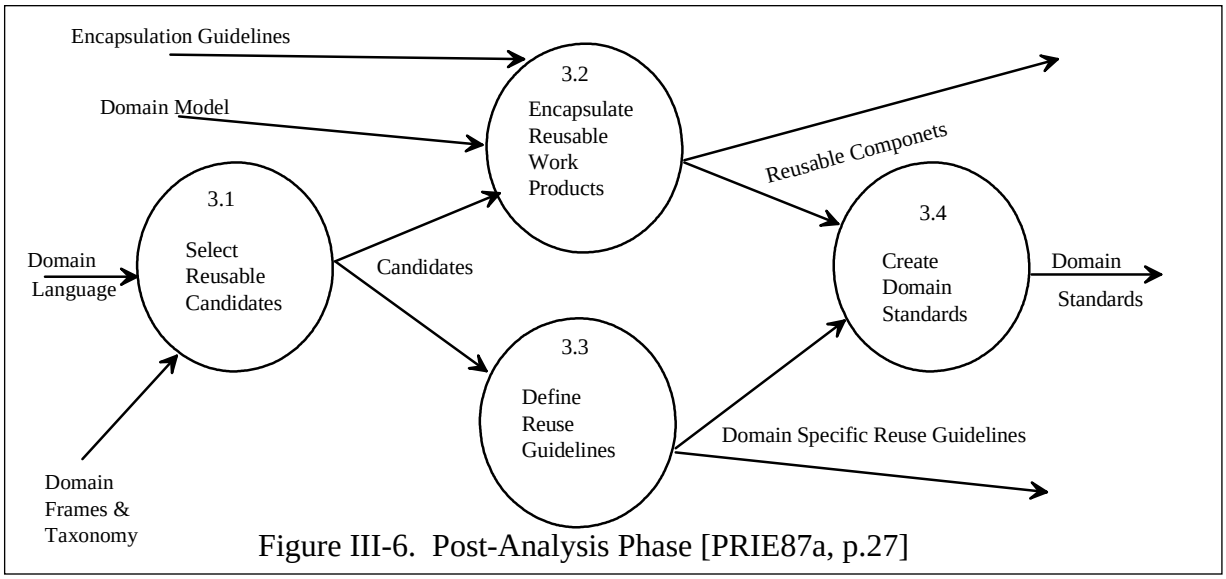


Figure III-6. Post-Analysis Phase [PRIE87a, p.27]

Prieto-Diaz appears to conceive domain analysis as a linear transformation that captures and represents domain knowledge in a model that enables generation of reusable components and creation of standards for reusing components. All information created during this transformation is then available to assist in engineering requirements for specific applications within the domain.

The approach outlined by Prieto-Diaz is unrealistic regarding several issues. First, understanding a domain sufficiently to generate a domain model is likely to be an iterative process. Management is unlikely to invest in an expensive, iterative activity as a front-end to requirements engineering. Further, application projects are unlikely to be help in abeyance, pending completion of a rigorous domain model. Second, creation of unique languages for each domain will likely inhibit integration between models from several distinct, yet related, domains. Such inhibitions will lessen the return from investing in the domain analysis. These limitations can be overcome, possibly, by incorporating domain analysis and modeling activities into the requirements engineering process outlined earlier.

Transforms 1.1 through 1.4 of Prieto-Diaz's pre-analysis phase (Figure III-4) map nicely into the elicitation step in the requirements engineering process. The inputs, for example, are identical: domain experts, existing systems, and analysis guidelines. The emphasis while analyzing a domain, rather than one specific application, would be broader and more general. Still, many of the processes and activities would coincide, and some of the same tools and techniques could support elicitation of knowledge about specific applications, as well as, knowledge about the entire domain in which the applications operate. Domain knowledge elicited could be used to better understand specific applications; and, because a number of applications are likely to be developed over time, two conditions might hold: 1) not all domain knowledge need be devised at once (refinements and additions can be made with each new application development) and 2) any domain knowledge previously acquired can assist in the elicitation, organization, and analysis of new applications in the domain.

Transform 1.5 of Prieto-Diaz's pre-analysis phase can be mapped to the organization step and the early portions of the analysis step defined for the requirements engineering process. Many of the tools and techniques available to help organize and analyze software requirements should also be applicable to defining domain requirements.

The analysis phase defined by Prieto-Diaz (Figure III-5) can be viewed as the later portion of the analysis step in the requirements engineering process. During this phase, domain concepts, functions, and relationships are identified and classified. The output of this phase includes a domain model, supported by a description of domain concepts (as frames) and relationships (as a taxonomy). Prieto-Diaz also proposes that during this phase a domain specific language be created; however, the current paper proposes that the domain specific

language in Prieto-Diaz's model be replaced with a representation that supports prototyping, and that steps 2.7 through 3.3 of Prieto-Diaz's model be replaced with a prototyping step. The aim of the prototyping step is to evaluate the domain concepts, functions, and relationships defined during organization and analysis. Is the understanding sound? Are the concepts reusable? Can the concepts be integrated with reusable components from other domains? Prototyping can help answer these questions, and a prototyping system can define an architecture into which reusable concepts and components can be fitted. Prototyping, in effect, can bridge the analysis phase to the post-analysis phase of Prieto-Diaz's domain analysis model by providing a means to evaluate domain knowledge, defined and represented during earlier phases, and to evaluate candidates for reuse. From here, the final step (3.4) of Prieto-Diaz's model can be followed. This step maps to the documentation step in the requirements engineering process; but instead of an SRS, the output includes: descriptions of reusable concepts and components, domain-specific guidelines for reuse, and domain standards.

Having described a mapping between domain analysis and a requirements engineering process, the current paper now considers tools and techniques that can support the integrated process. Specifically, techniques and tools for knowledge acquisition and representation are discussed in the next section, *Domain Knowledge Acquisition and Representation*, and prototyping approaches and systems are explored in Section V, *Prototyping and Reuse*.

IV. Domain Knowledge Acquisition and Representation

Among the most common problems found during a comprehensive study of a typical, large-scale software development project was a lack of widespread domain knowledge among the project team. [CURT88] Lack of domain knowledge was keenly felt in this large-scale project because,

> ... [a]lthough individual staff members understood different components of the application, deep integration of various knowledge domains required to integrate the design of a large, complex system was a scarcer attribute. Specification mistakes often occurred when designers did not have sufficient application knowledge to interpret the customers intentions from the requirements statement. As one system engineer put it: 'Writing code isn't the problem, understanding the problem is the problem.' [CURT88, p. 1271]

This same study also found that superior software system designers possessed a detailed understanding of the application domain, and could map between application requirements and the software structures needed to implement the requirements. From this study, three implications were drawn: 1) software tools and practices must raise the level of application domain knowledge across the entire development staff, 2) software tools and methods must accommodate experimentation and change, and 3) any software development environment must be a medium for communication. These implications lead directly toward methods for acquiring, representing, and sharing domain knowledge, and toward prototyping approaches.

This section of the paper considers techniques for knowledge acquisition and representation. The discussion begins by identifying sources for domain knowledge and describing some

of the problems faced by domain analysts (or knowledge engineers) when dealing with knowledge sources.

Three fundamental knowledge sources exist: automated systems that operate according to domain rules, technical documentation, and people who embody domain knowledge (so-called domain experts). A domain analyst can certainly sit down at a console and use an existing system to obtain direct, concrete, and unambiguous knowledge about an application. This is, perhaps, tedious, especially if the system is large. The domain analyst will probably still need to refine the knowledge gained from using the system by consulting with technical documentation and domain experts, but the method is well understood. Less well understood (as discussed earlier on pages 21 to 22) are methods to automatically extract domain knowledge from the actual code for the application system.

Review of technical documentation presents another set of problems. Reading a text on any subject requires that the reader possess a large base of knowledge, including general knowledge, subject-related background knowledge, and knowledge about how information is represented. [KONT88] These requirements place a certain burden on the domain analyst who must read and understand technical documents relating to an application domain. These requirements also impede the development of automated systems intended to extract knowledge from texts. An additional impediment to automated knowledge extraction is the tendency to rely on graphs, pictures, and charts in technical documents. Automated approaches for extracting knowledge from graphical material are beyond reach at the present time. [NAGY92] So, it appears that domain analysts must rely on domain experts to provide the bulk of domain-specific knowledge needed to understand particular applications.

Unfortunately, eliciting information from domain experts is difficult for a number of reasons. One problem, sometimes easily overlooked, is that many people who appear to be domain experts are, in fact, actually not. [CHOR90] Chorafas offers a set of guidelines to help knowledge engineers discern a real expert: 1) a real expert does not fear change, but understands that change is inevitable; 2) a real expert knows his own strengths and weaknesses; 3) a real expert appreciates gray areas; 4) a real expert can handle, and even thrive under, stress; 5) a real expert sees learning as a life-long endeavor and never misses an opportunity to extend her own knowledge; 6) a real expert sees sharing knowledge with others as a duty. [CHOR90, p. 43.]

After a domain expert is identified, other problems must be overcome by the analyst who hopes to acquire domain knowledge through interviewing techniques. The most basic problem is miscommunication between the knowledge engineer and the domain expert. [MUSE89] These two individuals rarely speak the same technical language. To reduce misunderstandings, the knowledge engineer must gain a prior knowledge of the application domain. Still, communication remains difficult because the analyst may need precise information, while the domain expert talks in inconsistent and imprecise terms.

Other problems relate to the nature of experts. For example, experts do not introspect reliably. [MUSE89] Experts are usually bad at explaining, but good at doing. [CHOR90] Experts have difficulty articulating problem solving knowledge in a form suited for representation in an expert system. [BOOS86] For these reasons, the domain analyst must find an expert who can, and who is willing to, spend a few hours exploring a domain, and who can be on call while the elicited knowledge is encoded in an expert system and then tested. [CHOR90]

Even more fundamental problems face the domain analyst because the model provided by expert systems (i.e., a knowledge base, coupled to an inference engine) does not seem to reflect the way experts actually think. [MUSE89] In experts, changes in the knowledge base seem to alter thought processes. For example, as people move from novice to expert, they tend to chunk knowledge into highly specialized, content-specific and task-specific methods. Experts rely on a huge amount of this specific, content knowledge. The process knowledge used by an expert then becomes dependent on the content knowledge possessed by that expert. Expert systems are not presently capable of altering their reasoning methods as they obtain increased content knowledge over time. This difference creates an incompatibility between the basic model used by a knowledge-based system and the thought processes of experts.

Although these problems represent barriers to a domain analyst, research into knowledge acquisition techniques suggests some strategies, techniques, and tools that can help overcome such barriers. Knowledge acquisition research considers both human techniques and automated methods. These are discussed in turn.

The technique most often used by domain analysts to elicit knowledge from domain experts is the interview. [CHOR90] In general, when dealing with an expert, an interview is one-on-one. The domain analyst must have an understanding of the concepts and jargon in the domain and must know what questions to ask. Both the interviewer and the expert must see the interview as an iterative process. The results of interviews alone are usually unsatisfactory, so interviews are often coupled with observation of the expert at work. The intent of such observation is to verify and clarify information obtained during interviews, as well as to prepare additional questions for future interviews.

Other strategies can augment interviews, and, thus, strengthen the knowledge acquisition process. [CHOR90] For example, the domain analyst can set up a recorded session where the domain expert, without the domain analyst present, is introducing two novices to the problem domain. The presence of two novices raises the odds that the session will be interactive. The domain analyst can use the recording to prepare for interviews with the domain expert.

Another technique available to a domain analyst is the expert workshop. [CHOR90] In this technique, the analyst conducts a workshop with the domain expert where case studies and scenarios are presented to the expert. The analyst actively probes with questions to reveal the experts reasoning, and takes readable notes for later review. This technique can be strengthened by sampling several experts in an N-FOLD approach. While interviewing multiple experts increases the cost of knowledge acquisition, application of N-FOLD techniques to other front-end software development activities appears to be cost effective. [MART90] A more difficult problem when sampling experts is that domain knowledge is not always additive; in fact, an analyst might be forced to select between divergent approaches or rules gleaned from different experts.

While interview techniques for knowledge acquisition remain largely a human endeavor, many researchers aim to improve the efficiency and effectiveness of interviewing by employing automation. Two general approaches are being pursued: 1) automated assistance for the domain analyst and 2) automated elicitation, directly from the domain expert. The automated assistant approach appears generally applicable to a full range of problems, from small to large, across many domains. The focus in these approaches is to help an analyst by exploiting the strengths of automation. The direct elicitation approach appears more limited by a need for a priori, domain-specific

knowledge. To overcome this limitation most research on direct elicitation provides tools that could be viewed as expert system generators, that is, for each specific domain, a computer program elicits information from a domain expert and then generates an expert system for the domain. Of course, some researchers are investigating hybrid approaches to provide knowledge system workbenchs that provide tools for automated assistance of analysts, as well as tools for automated generation of expert systems. Some specific tools supporting each of these approaches are described below.

Fickas envisions a tool, KATE, for automating the analysis process. [FICK87] Fickas views the requirements analysis process as an interactive, iterative, problem-solving paradigm involving a user and analyst. To support this paradigm, Fickas plans KATE to consist of a front-end (which includes a domain model, a knowledge acquisition component, and a critic) and a specification generator (which can map from an internal knowledge representation into any number of existing specification languages). KATE's domain model would encompass a wide range of knowledge: 1) common objects, operations, and constraints, 2) known solutions to hard design and implementation problems, 3) an understanding of how the environment might affect the system, and 4) a model of typicality for the domain, but modulated by any management policies. Such a model is easier described than implemented.

At the time Fickas was reporting his approach, he had implemented a small-KATE (SKATE). The main purpose of SKATE was to show that domain knowledge can be used to detect errors in a problem description that cannot be detected solely using syntactic knowledge. The domain model constructed for SKATE was hand-coded using Knowledge Engineering Environment (KEE), an expert system shell that features a frame-oriented representation language (see the discussion of knowledge

representation below). The main result of SKATE was learning about what would be required to make KATE a success. The domain model must be more general, including support for learning new concepts as they arise. KATE must embody enough domain knowledge to avoid requiring the user to input the tedious lists of objects, actions, and constraints for the domain. KATE must also include an example generator, because a strength among domain experts is the ability to generate significant examples that reveal domain rules that might otherwise be overlooked.

Another component of Fickas' paradigm is the critic. The job of the critic is to poke holes in a requirements description by using the domain knowledge base from KATE. For the exercise with SKATE, Fickas implemented a critic called JOG. In experiments with a library problem domain, JOG was able, while analyzing a requirements statement, to discover: missing resource classes, lack of a borrowing time limit, a missing concept of books on reserve, missing limits to the number of books that could be checked out at one time, omission of security requirements, and missing queries that a borrower might wish to make. While this performance was impressive, success depended largely on an extensive domain knowledge base which was hand-coded by a human knowledge engineer after interviewing domain experts. As Fickas pointed out "... we have yet to demonstrate an interactive acquisition model, nor a means for learning new concepts introduced by a client." [FICK87, p. 66] In effect, all SKATE and JOG accomplished was to demonstrate the type of automated assistance that could be provided to a requirements analyst, given a sufficient domain knowledge base.

Another researcher, Shemer, describes a systems analysis expert aide (SYS-AIDE) intended to help an analyst interview and collect assertions about the problem domain and to construct a conceptual model. [SHEM87] SYS-AIDE is modeled as an expert system asking questions in order to build a model. The

questions asked by SYS-AIDE should point the analyst to the information that must be collected to successfully construct a model of the domain. In effect, SYS-AIDE elicits assertions from the analyst (who must then elicit knowledge from a domain expert) and, given the assertions elicited, maintains a conceptual model by adding and modifying assertions, and by reasoning about relationships. The specific assertions vary with the domain, but the reasoning rules are invariant knowledge encoded within SYS-AIDE. In effect, the invariant knowledge is analysis process knowledge, so SYS-AIDE attempts to be an expert domain analyst.

Another tool to assist the domain analyst is reported by O'Bannon. [OBAN87] This tool is similar in concept to SYS-AIDE, but adds knowledge of elicitation strategies. A rule constructor encodes methods to elicit, record, and analyze responses from a domain analyst. During knowledge acquisition, the system offers the domain analyst a prioritized list of actions to be taken at each interview step and suggests the most effective verbal responses to elicit the necessary information. Once the information is entered into the rule constructor by the domain analyst, the system can analyze the logical implications of the assertions and produce a preliminary set of production rules.

Another approach built on the principle of assisting the analyst is described by Palmer and Fields of George Mason University (GMU). [PALM92] The environment developed at GMU covers the entire requirements engineering process, not simply knowledge elicitation; however, tools are included for eliciting requirements from groups of users and for analyzing requirements for conflict, redundancy, incompleteness, imprecision, and ambiguity. The environment provides a nice framework, as well as a set of useful tools. Plans for future work include incorporating more semantic knowledge into the analysis tools.

Since the environment includes capabilities for prototyping, a possibility exists to investigate means of tying domain models to prototypes through specific knowledge representation schemes. The requirements engineering research at GMU goes further than most other efforts to establish an environment into which tools can be fitted. Such an environment enhances the effectiveness of individual tools by enabling them to be combined and applied together in novel ways.

A knowledge acquisition tool that seems to fit within the philosophy of the GMU requirements engineering environment is REMAP. [RAME92] The aim of REMAP is to capture deliberations that occur during requirements engineering and software design. Specifically, REMAP captures the design rationale so that the design is more likely to be understood later, and, therefore, reused. REMAP supports the incremental and iterative nature of the requirements and design processes. Perhaps REMAP could provide a bridge between the GMU requirements engineering environment and the software design process.

Another automated assistant that attempts to bridge between requirements and design is Fickas' Critter. [FICK92] Critter embodies knowledge of system design strategies and concepts; a human designer is expected to provide domain knowledge. Critter and the designer interact to develop a design to solve a domain-specific problem. To date, the results with Critter are not encouraging. Critter's limited reasoning techniques prevent its use on large software engineering problems; the analysis algorithms used in Critter are much too slow for an interactive design system; and Critter's knowledge base and representation omit several classes of system design concepts.

Another class of knowledge-based systems attempts to replace the domain analyst with an expert system generator. Such generators interact directly with a domain expert to create an expert system for a specific domain. One such system, the

Expertise Transfer System (ETS) reported by Boose, attempts to cut the time required (typically six to 24 months depending on the domain) to create an expert system using interview techniques. [BOOS86] ETS employs clinical psychotherapeutic interviewing methods, called Personal Construct Theory, originally developed by George Kelly. [KELL55] ETS automatically interviews an expert, analyzes the information gathered, and then generates a set of production rules. ETS incorporates an inference engine to permit testing of the generated production rules.

ETS first elicits conclusions that the expert system to be generated should be reaching (e.g., specific diseases, management decisions, diagnostic recommendations). If the expert does not know, an incremental interview mode is started. Once the conclusions are captured, ETS presents these, three at a time, to the expert and inquires about similarities and differences. The result of this initial phase is a list of elements to be classified and a list of classification parameters. During a second phase, the expert is asked to rate each element against each pair of traits using a numerical scale, augmented with the ratings neither and both.

Once a rating grid is established, ETS invokes several analysis methods to structure the knowledge. First, ETS builds a graph of implied relationships, and then computes matching scores between traits and goals. The expert is consulted to help distinguish between closely related concepts. Second, ETS generates production rules of two types: conclusion rules and intermediate rules. Each rule is associated with a certainty factor between -1.0 (False) and +1.0 (True). Once production rules are generated, the expert can use the inference engine in ETS to test the knowledge base, or she can generate a knowledge base for input to an expert system shell, such as KEE.

ETS, while a powerful tool, is not without limitations. For example, ETS is best suited for analysis class problems where the solutions can be enumerated ahead of time. ETS cannot handle constructive-class problems where unique solutions are built from components. Also, the grid method cannot elicit deep causal knowledge, procedural knowledge, or strategic knowledge. For some domains, the expert finds it difficult to identify similar sets of conclusions at useful levels of granularity. Experts can also have difficulty interpreting the meaning of certainty factors, when viewed in isolation, because these factors are relative to each other. Some practical problems are also annoying: knowledge grids developed by individual experts cannot be combined and knowledge grids cannot be updated easily.

Another system, PROTEGE, reported in the literature is similar in approach to ETS. [MUSE89] PROTEGE requires that a knowledge engineer provide general knowledge about an application area. With such knowledge, PROTEGE can generate knowledge editors for the application area. Domain experts then use the knowledge editor to produce specific expert systems for the application area.

PROTEGE and ETS are representative of a class of systems aimed at generating expert systems, either from experts or from an existing knowledge base. Other examples of such systems include: META-DENDRIL [BUCH78], AQ11 [MICH80], TEIRESIAS [DAVI81], and NANOKLAUS [HASS83]. Some researchers have picked up this trail and are attempting to employ expert systems to support domain-specific reuse.

Iscoe advocates using expert systems to enable users "... to directly create and maintain their own programs because it is unreasonable to expect that there will ever be enough professional programmers to meet the continually increasing demands for software." [ISCO88, p. 301] Iscoe believes that domain knowledge is the key to successful user programming. He

outlines a research program to explore methods to build a domain model as a knowledge base, and then to use the knowledge to transform user-provided, application specifications into run-time programs. As with most approaches of this type, Iscoe intends to limit his investigation to programming in a small domain.

Another interesting approach, still limited to a narrow domain, attempts to generate knowledge directly and automatically from technical text. [KONT88] Kontas and Cavouras propose to parse text from a technical document to create an attribute grammar (both syntax and semantics) representation of the information. Their research works with a BASIC programming document as the input text. They have determined that parsing the text requires: subject-related knowledge (i.e., BASIC programming concepts), linguistic knowledge (i.e., lexical, syntactic, and semantic knowledge of English), and representational knowledge (i.e., attribute grammar rules). While this approach appears unique within the literature, the results are not encouraging. Parsing is slow; the subject text is very narrow; attribute grammars can only represent two types of relationships; the vocabulary processed is small; and the resulting representation can only answer a single type of question.

More promising than any of the approaches that replace domain analysts with expert systems, hybrid approaches incorporate automated assistants together with expert system generators and other supporting tools to create a knowledge acquisition workbench. A typical example of this hybrid approach is AQUINAS. [KITT87] AQUINAS, shown in Figure IV-1, gives automated assistance to both domain analysts and domain

experts.

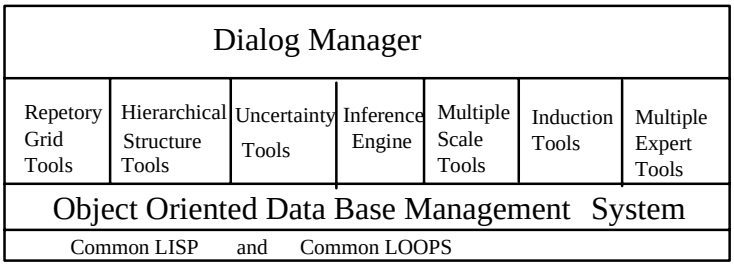


Figure IV-1. AQUINAS Knowledge Acquisition Workbench [KITT87, p. 99]

Tools from the AQUINAS workbench help analyze the problem domain, classify the problem task (and any subtasks), identify problem solving methods, suggest appropriate knowledge acquisition tools, and recommend specific strategies for using the tools. In philosophy, AQUINAS is similar to the requirements engineering environment developed at GMU. A range of tools is included inside an integrated environment; analysts can use the tools in a variety of novel ways; and AQUINAS even advises analysts on appropriate methods to acquire knowledge.

In the preceding discussion of knowledge acquisition techniques, methods of representing knowledge within a knowledge base were sometimes mentioned. A more complete presentation is in order because knowledge representation is the single most important factor determining the power of expert systems applications that can be built. [CHOR90]

Knowledge representation schemes can be divided into four categories: logic programming, production rules, frames, and semantic networks. These schemes are usually mixed within practical applications of knowledge-based systems, but considering each representation in isolation provides an appreciation of the strengths, weaknesses, and applicability of each.

Logic programming is programming by description. [GENE85] Specific assertions are declared explicitly to be true, and then are combined with a general inference procedure. The assertions describe objects and relationships within a problem domain. The inference process requires that each statement in the knowledge base must be capable of evaluating to true or false. In effect, execution of the program is modeled as a deductive proof. Logic programming permits incremental development of the knowledge base and enables the program to explain how it solved each problem, and to tell why it believes its answer to be correct. The practicality of logic programming in software engineering depends to some extent on the underlying technology. In general, logic programming systems are inefficient and difficult to use. Another drawback of logic programming systems is the requirement for evaluation as true or false. While such evaluations are necessary to deductive methods, many expert systems must draw conclusions from uncertain data, must reason by analogy, and must generalize from existing knowledge.

A derivative of logic programming, rule-based systems, represents knowledge as a set of production rules. [HAYE85] Production rules are "if-then" constructions that can combine preconditions using Boolean operators. When the "if" clause evaluates to true, the "then" actions can be executed. The skill of a rule-based system increases proportionally to the size of the rule set. Expert systems have solved a wide range of complex problems by selecting and evaluating production rules. Such systems can select adaptively the best sequence of rules to use, and can explain the conclusions reached.

As successful as rule-based systems have proved in practice, a number of problems limit their potential. At present, there is no analytical foundation for rule-based systems that allows separating solvable problems, from unsolvable ones. Also, suitable techniques are needed to test a

rule set for consistency and completeness. In practice, rule sets lead to slow executions and are difficult to scale up to sizes required for large problems. Finally, methods are needed to seamlessly integrate rule-based systems into normal data processing applications.

A different school of knowledge representation deemphasizes logic programming and production rules in favor of descriptive templates, called frames. [FIKE85] Frames provide a rich structural language for describing domain objects and some basic relationships between those objects. For a given object, attributes and taxonomies can be represented. Within some frame representations, action procedures can be attached to specific attributes. The concept of frames matches nicely the theory of object oriented programming.

Reasoning services available in a pure frames representation are limited to the taxonomies between the frames, usually this means inheritance relationships, attribute value groupings, and cardinal relationships. Sometimes, frames are conceived as the database of a knowledge base; in such conceptions, frames are normally augmented with production rules, or some other method of describing relationships, to form a hybrid representation (see, for example, KEE).

In an effort to enhance the value of frames, some researchers propose using semantic networks. [EEPE92] Semantic networks provide natural ways of representing inheritance and aggregation. Searching through the frame structure based on inheritance and aggregation relationships or for exact matches among sets of attribute values is very efficient, but these are the only types of searches that semantic networks allow.

Even in the case of inheritance relationships, semantic networks leave two situations unaddressed. One is exceptions to a classification hierarchy. For example, if elephant is a class with the attribute color = gray and royal_elephant is a subclass

of elephant with the attribute color redefined to equal not_gray, then, if Joe is a royal_elephant, is Joe gray? This is an ambiguous question in a semantic network. Normally, exceptions are resolved by using the traits of the nearest relative, so the ambiguous question would be answered: no. The second situation not addressed by semantic networks is multiple inheritance. For example, consider two classes: Buddhists and athletes. Buddhists are vegetarians and athletes are non-vegetarians. If Tom is a Buddhist and an athlete, then is Tom a vegetarian? Proposals exist to solve multiple inheritance problems in semantic networks through evidential reasoning techniques. For example, relative frequencies can be encoded into attributes in each frame, and then reasoning can be based on probabilities. The combination of frames and semantic networks appears promising as a knowledge representation method, but more research is needed.

Even as available today, frames, coupled with semantic networks, have some advantages over logic-based approaches. For example, frames allow knowledge to be represented in a form that experts typically use; semantic networks allow concise structures to represent certain relationships (i.e., inheritance and aggregation); frame-based, semantic networks support specialization from more general concepts; and semantic networks of frames enable construction of special-purpose, high-performance, deductive algorithms. However, without augmentation by production rules, the ability to reason about frames is strictly limited to the relationships and values encoded in the frame structure.

Practical knowledge-based systems intended to support development of large, software systems will require multiple kinds of knowledge, represented in the most appropriate form for the intended application. Rich and Feldman describe such a system, called CAKE. [RICH92] CAKE is a prototype knowledge

representation and reasoning system used to generate two automated assistants for programmers: the Requirements Apprentice and the Debugging Assistant. CAKE enables representation of structural artifacts (e.g., specifications, programs, requirements) at various levels of abstraction. CAKE evaluates the reasonableness of decisions reached by a program, fills in missing details, consults with the programmer before committing to complex decisions, and explains to the programmer the actions taken and decisions made. Providing these capabilities requires seven levels of knowledge within CAKE (see Figure IV-2).

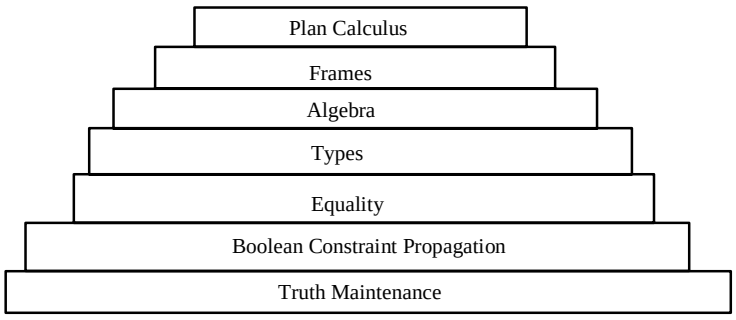


Figure IV-2. Seven Layers of Knowledge and Reasoning Within CAKE [RICH92, p. 452]

Notice that the knowledge representations encompassed by CAKE include logic programming, production rules, frames with semantic networks, and more.

While the effort behind CAKE is impressive, the tool is not very usable, except by the most knowledgeable programmers. Also, the knowledge structure cannot be updated easily in some cases. As first developed, every fact in CAKE was retractable so that incremental development could be supported. Unfortunately, the resulting system was too slow. Now, two parallel mechanisms are implemented: one efficient, but non-retractable, the other, more expensive, but retractable.

Before concluding this discussion about knowledge acquisition and representation, consideration of problem solving

strategies is in order. The problem solving strategy used by a specific expert system defines the types of problems that the system can solve. This is an important consideration because software reuse falls into a particular category of problems that most expert systems are not equipped to solve.

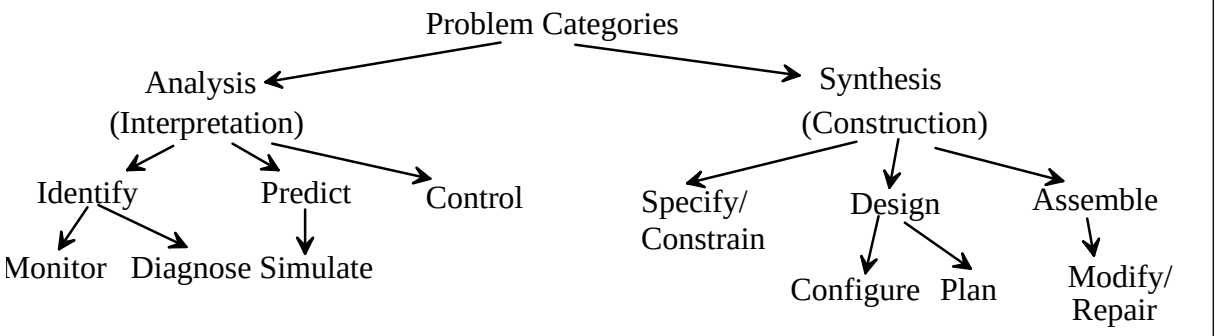


Figure IV-3. Problem Categories for Expert Systems [KITT87, p. 97]

Figure IV-3 presents a taxonomy of problems that expert systems might help solve. Analysis problems require selection of the best solution from among a set of known solutions. [CLAN86] Synthesis problems require composition or configuration or modification of components to construct a previously unknown solution. The majority of expert systems employ heuristic classification strategies that apply to analysis problems. A small minority of expert systems use heuristic composition methods that apply to synthesis problems. Unfortunately, reusing software from a component library, given some instructions from a user and some knowledge within an expert system, presents a rich set of construction-type problems. How, then, can domain knowledge represented within expert systems be applied to requirements engineering and software reuse?

Domain knowledge, encoded in expert systems, can be used to analyze requirements statements for amguity, incompleteness, and conflict. Such reuse of domain knowledge during requirements engineering has been demonstrated in research results such as SKATE. Automated elicitation and subsequent representation of

domain knowledge in a form that can be applied to requirements analysis appears feasible because expert systems have a history of success when applied to analysis and classification problems.

Elicitation and representation of domain knowledge in a form that facilitates construction or synthesis of software systems from reusable components is an open question. To accomplish such results, knowledge of many domains must be represented in a form that can be interpreted and acted on by a software constructor or program generator. In effect, successful creation of an application program from reusable software must use knowledge to build a bridge between requirements and code. The semantic gap between these levels appears large. A prototyping step within the requirements engineering process presents an early, inexpensive opportunity to demonstrate the feasibility of automatic program construction or generation. The next section of this paper considers connections between domain knowledge, prototyping, and software reuse.

V. Prototyping and Reuse

The idea of prototyping for software systems originated from an observation: when a programmers build software to solve a problem with which they are unfamiliar, the first implementation is almost always inadequate, but the second is adequate. Sometimes, the original development is not delivered to the customer, but contributes to an extended schedule and to cost overruns. Some practitioners and researchers conceived of a development process that takes advantage of these early development efforts by sharing the results with users in order to obtain feedback regarding requirements and preferences.

Since these initial developments, prototyping concepts have matured greatly. In fact, there are now various theories and classification schemes for prototyping approaches. For example, Sage and Palmer divide prototyping techniques into three categories according to goal, or aim: 1) purposeful, 2) functional, and 3) structural. [SAGE90] According to Sage and Palmer, purposeful prototyping focuses on verifying user requirements and ensuring that software requirements are consistent; functional prototyping intends to verify that a system will accomplish what the user wants; and structural prototyping tests the feasibility of a design approach. Sage and Palmer also identify six other classification schemes proposed by researchers and provide a mapping between these schemes and their own proposals.

Sage and Palmer share some other insights that should be kept in mind when evaluating prototyping techniques. For example, although many users argue that iterative prototyping is too expensive, Sage and Palmer point out that finding errors during the requirements phase means corrections will be much, much cheaper than if the same errors were not discovered until

later in the development process. Sage and Palmer also indicate that not all application domains are amenable to prototyping. They state that real-time systems are not good candidates for prototyping, that large systems can only be prototyped in a limited fashion (perhaps to identify user requirements), that small systems can be prototyped to experiment with the user interface, and that conventional, information systems might be appropriate candidates for full prototyping.

In the current paper, prototyping approaches are classified in two ways: by technical approach and by life-cycle model. Different ideas are discussed for each of these categories, and specific examples are described. Two ambitious prototyping research projects, the Programmer's Apprentice [RICH88] and Easy Programming [MARQ92] are singled out for more detailed consideration. The section closes with a discussion of relationships between prototyping and reuse.

Three technical approaches to software prototyping can be discerned from the literature: transformation, composition, and simulation. The vision of advocates of the transformation approach, often called automatic programming, was described in the literature in 1983 by Balzer. [BALZ83] Balzer's vision of automatic programming was refined in a 1985 article by Barstow.

> An automatic programming system allows a computationally naive user to describe problems using natural terms and concepts of a domain with informality, imprecision and omission of details. An automatic programming system produces programs that run on real data to effect useful computations and that are reliable and efficient enough for routine use. [BAR85, p. 1321]

This definition of automatic programming provides a tall order: Barstow saw automatic programming not as a prototyping technique, but as an approach to operational software

development. Barstow recorded the difficulty of achieving his vision of an automatic programming system (APS): an APS requires a great deal of domain knowledge (definitions, problem-solving heuristics, and expectations about run-time characteristics of the data). Without such knowledge, a computationally naive user will have difficulty expressing herself to the APS. Barstow foresaw a user writing an informal specification that would be transformed by the APS through a series of steps to become one or more programs.

Probably the most well known transformation system reported in the literature is DRACO. [NEIG89] (Although Neighbors described DRACO as a hybrid between transformation and composition, DRACO performs a serious of transformations between multiple levels of abstractions.) DRACO consists of: 1) a library of domain-specific notations, each narrow in scope, but not hierarchically organized, 2) a parser, 3) a pretty-printer, 4) generators, and 5) analyzers. The basic approach to programming in DRACO is to use domain analysis to develop a Domain Language and then to parse the language and input the parse-trees to a code generator. The semantics of a component in one domain are defined by translation into components of other domains; thus, the transformations comprise a hierarchy of domain models. Neighbors describes the execution model as well understood, and he believes that mapping between domain models and the execution model is straightforward. He identifies the main problem with the DRACO approach to be construction of models for application domains. In other words, who will construct the knowledge base needed to describe the application?

A second technical approach to prototyping relies more directly on composition from reusable components. An example of composition, called MELD, is described by Kaiser and Garlan. [KAIS89, GARL92] MELD defines a module interconnection language that can be used by an implementor to compose an environment

from a collection of features, each of which implement some basic functionality. MELD assumes that a set of reusable features have been encapsulated into a library. Composition as described with MELD can only be achieved by professional programmers, but the potential to quickly construct prototypes is intriguing. The initial problem, of course, is construction of a library of reusable features.

A third technical approach to prototyping provides a simulation of system behavior. Lee and Sluizer describe a language, called SXL, that allows system behavior to be modeled as a finite state machine, with pre- and post-conditions and invariants included for each transition. [LEE91] SXL descriptions, based on entity-relationship structures and quantified, first-order logic, can by executed interactively to test the behavior of a system. An earlier simulation approach to prototyping was described by Zave. [ZAVE84] Operational simulations can only be used to evaluate system behavior. Issues such as the user interface, data requirements, and communications interoperability cannot be evaluated through operational simulation. Also, prototyping via simulation is best performed by a trained analyst or programmer.

A third way to view prototyping approaches is by life-cycle model: throwaway, evolutionary, or operational. Each of these is considered in turn.

Throwaway prototypes are intended to verify some specific, but limited aspect of system requirements, such as complex behavioral requirements, user interface requirements, or requirements for an innovative algorithm. One advocate of throwaway prototyping is Andriole. [ANDR92] Andriole proposes a storyboard approach to prototype user interface requirements.

> Storyboard designers must design every single display of the system in sequence, with explanations and descriptions of each display. All methods and algorithms should

> be explained and users should be able to get a solid feel for how the system will work just by thumbing through the pages. [ANDR92, p. 11]

The availability of high-quality, inexpensive graphic displays and graphic design software make Andriole's approach particularly attractive because users can get a computer look-and-feel for the system interface without software development. Also, modifying the display screens becomes a quick, simple job.

Another approach to quick, throwaway prototyping is under investigation at Oregon State University. [LEWI89] Lewis and his colleagues are developing a prototyping system that maps between a user interface and a set of actions. In essence, the system proposed by Lewis, augments graphical storyboards with a set of actions that can switch between displays, pull down menus, and perform some limited processing from scripts input by a user. A user or programmer can interactively design displays and menus, can assign behavior to specific user actions, and can create scripts defining operational functions. The tools under construction to support this project appear promising as a quick prototyping method.

Evolutionary prototyping, as opposed to throwaway approaches, constructs an application system incrementally, refining each increment through interactions with the users, until an operationally complete system is constructed. The prototyping efforts are meant as a lasting investment. In 1983, Balzer described an approach to automatic programming that, in effect, envisioned evolutionary programming.

A concrete evolutionary software development system, the computer aided prototyping system (CAPS), is described by Luqi. [LUQI89] CAPS comprises three subsystems: a user interface, a software components database, and execution support. The user interface subsystem provides two editors (one syntax-directed

and the other graphical) for defining specifications and user interface screens, an expert system to help users create specifications, and an assortment of debugging and browsing tools. The software components database subsystem provides tools to manage a repository of design descriptions and translation rules. The execution support subsystem consists of a translator and two schedulers (one static, one dynamic). The user specifies an application using a Prototype System Description Language (PSDL). The translator converts the specification into an executable system, drawing on previously defined components stored in the repository. The user may then exercise the prototype under control of a run-time scheduler. CAPS is evolutionary because specifications written, translated, and tested can be retained in a database for use by prototypes developed later.

Recently, Davis proposed combining throwaway and evolutionary prototyping to form a third approach that he names operational prototyping. [DAVI92] Operational prototyping calls for layering rapid prototypes on a solid, evolutionary base. In Davis' view, evolutionary prototypes are built with quality by following a conventional software development approach. In fact, only confirmed requirements are implemented in an evolutionary prototype. Quick prototypes are implemented to explore poorly understood requirements, then discarded. Davis imagines building rapid prototypes within an architecture that embodies an evolutionary prototype. Davis believes that the key to achieving an operational model is to build evolutionary prototypes within an architecture that accommodates extensive change. Although Davis does not describe such an architecture, others have.

Holt and Stanhope define an architecture to support operational prototyping. [HOLT91] The architecture proposed by Holt and Stanhope builds on a set of reusable objects, standard

interfaces to those objects, standard interfaces for communicating between computers, and a set of tools for specifying, composing, and executing applications. The reusable *megaobjects* proposed by Holt and Stanhope differ from the small objects usually associated with object-oriented programming.

> 'Megaobjects' are large pieces of software that contain carefully defined and encapsulated interfaces. During the 1990's, applications that are well understood will be captured as megaobjects. These will communicate by software buses, standardized mechanisms for communication. The principles and protocols for software buses are currently being formulated, and configuration systems for interconnecting megaobjects via a software bus without coding will become common. [HOLT91]

Examples of megaobjects already exist: X-Windows libraries, the Motif graphical user interface, TCP/IP software, SQL interface libraries for relational database systems, and various standards for interchanging formatted data (e.g., SGML, ODA/ODIF, IGES, and PDES). The concept of a software bus enabling megaobject interconnection without programming is, however, a bit mysterious. (Certainly, applications can be linked to known megaobjects via link libraries and data can be exchanged between loosely-coupled megaobjects using standards for data communication and formatting, but a grander vision of a high level composition language for megaobjects is not yet feasible.)

The architecture proposed by Holt and Stanhope meets the criteria set by Davis for operational prototyping: a solid evolutionary base, amenable to extensive change. In fact, Holt and Stanhope see their architecture as an operational model for computing based on incremental development by different classes of developers. They envision three types of software developers in the year 2000: end users (estimated to number 110 million) , application programmers (expected to number 1 million), and

toolsmiths and software engineers (expected to number 100,000). The software engineers will build the megaobjects and work on standards. The toolsmiths will incorporate megaobjects into higher level tools, such as parameter-driven form generators. End users can employ the generator tools to construct applications. To support their vision, Holt and Stanhope cite today's user-driven software systems.

> Most popular software tools ... available today are configurable objects that are parameter-driven. These include spreadsheets, databases, wordprocessors, and multi-media systems. All of these provide the capability to enter data and instructions on what to do with the data. The data and instructions are then read by the execution engine, and executed to produce the results desired by the user. If they don't match the user's needs, then the user can change the specifications using an intelligent, interactive graphical user interface. [HOLT, p.53]

Holt and Stanhope go so far as to set goals for the next two decades: by 2000, 50% of applications will be built with tools and by 2010, 75% of applications will be so built.

To better gauge the practical possibilities of prototyping as a development method (i.e., the evolutionary and operational approaches), a review of progress made toward knowledge-assisted programming is in order. In 1988, Rich and Waters reported on an ambitious project to provide computer assistance to programmers. [RICH88] This *Programmer's Apprentice* is intended to support all phases of software development from requirements analysis through software testing. The apprentice and programmer are to communicate through a body of shared knowledge about programming techniques.

Rich and Waters recognized that engineers think in chunked concepts, that they labeled clichés, and that these concepts usually related to one another. They decided that, given a

library of standard clichés and assistance from an expert system, programs could be constructed by inspection, rather than by reasoning from first principles. Defining and representing the necessary clichés became a major focus of the Programmer's Apprentice project. In effect, Rich and Waters were building a model for the domain of computer programming.

The programmer describes a specification to the apprentice through a formal notation, called a Plan Calculus. Using the programmer-provided plan and a previously encoded knowledge base, the Programmer's Apprentice can reason about the program and can map the plan to an implementation. Rich and Waters reported that the initial phase of the project concentrated on generating an implementation from a programmer-created design. Since that time, other apprentice tools (for example, a Requirement's Apprentice and a debugging assistant) have been produced to assist with other phases of the software development process.

The Programmer's Apprentice project provided effective tools that were difficult to use and performed poorly. Certainly, the Programmer's Apprentice could assist a patient, professional programmer, but providing help to a computer-naive user was beyond its capabilities. Four years later, in 1992, Marques and his colleagues at the Digital Equipment Corporation (DEC) described a knowledge-assisted system that is intended for easy programming by users.

In outline, the DEC system maps the features of a specific application to appropriate abstract methods (i.e., control structures stored in a knowledge base), elicits expertise (including variations and exceptions), translates the expertise into a form that the selected abstract control structure can use, and then modifies and extends the application to cover changes in the application requirements. To accomplish these tasks, the DEC system comprises three tools: Spark, Burn, and

Firefighter. For a better understanding of the system, each of these tools in discussed in turn.

Spark, with help from a user, sifts through a hierarchy of pre-defined control structures to select an appropriate approach for the specific application at hand, and then, by consulting with the user, customizes the selected approach. Each component in the hierarchy is characterized by a set of assumptions about the type of inputs needed and the kind of outputs produced. Where multiple control structures appear to be appropriate, Spark queries the user to reach some conclusion on which structure would be best. If Spark cannot easily explain the source of ambiguity to the user, then Spark simply makes some default assumptions and leaves the problem for Firefighter. After completing its work, Spark calls Burn to further customize the selected solution.

Burn relies on a library of knowledge acquisition tools, one is associated with each pre-defined, control structure. Each knowledge acquisition tool knows what knowledge is required for its associated control mechanism, knows how to elicit the needed knowledge, and knows how to represent that knowledge in a form needed by the control mechanism. For example, Burn might ask the user for some solutions to an example problem and for a means of distinguishing between the solutions. After Burn acquires the necessary knowledge and configures pull-down menus for the application, Firefighter is dispatched.

No program generated by Burn will work well until it has been used for a while, and is then modified to account for forgotten or unanticipated factors. Burn programs are executed under the control of Firefighter. Firefighter is an evaluator that monitors the performance of Burn programs, detects poor results, and then queries the user to diagnose and debug the application. If a detected error results from missing or incorrect knowledge, then the knowledge acquisition tool is

invoked. If the control mechanism is inappropriate, then Spark is invoked to select a new mechanism.

Firefighter employs three rather sophisticated, complementary evaluation techniques to monitor the performance of Burn programs. The first two evaluation techniques rely on specific code that is included in the control mechanisms, while the third technique is built into Firefighter. The first evaluation technique might be called: GOOD DOG, BAD DOG. Each time the application executes, the user is queried about whether the performance was adequate. If a BAD DOG response is received, then the knowledge acquisition tool is invoked. The second evaluation technique might be called: I'VE BEEN A BAD DOG. The application monitors its own performance to detect inconsistencies and inadequate results. When such problems are detected, the user is informed and the knowledge acquisition tool is invoked. This strategy is necessary because most users will not sit still during the initial development while Burn elicits knowledge about every type of case that the program might face. Instead, Burn asks for a minimum of information to start, the application then monitors its own performance, and the user is required to provide additional knowledge as needed to resolve problems and improve the performance of the application. The third evaluation strategy might be called: I THINK YOU MIGHT NEED A HORSE. Since Spark initially selects a control mechanism by making strong assumptions on weak evidence, Firefighter must compare the application output to the assumptions in order to detect incorrect control mechanisms. When an error is suspected, Spark is invoked to suggest an alternate control mechanism.

The goal of the DEC system is to supply reusable mechanisms in a usable fashion. Marques and his colleagues plan an elaborate set of steps to evaluate progress toward their goal. To assess usability they built nine applications themselves, and

then presented them to users. (At the time of the report, these applications were being evaluated by the users.) If the applications appear useful, they plan to write detailed instructions for specific application tasks and then to ask users with various levels of programming skill to build some programs to solve the tasks. Then, they will ask domain experts, who perform a task well, but manually, to create a full-scale program using the tools. (At the time of the report, one program had been built by a user; the job took eighteen days.) As a final test, they will ask an experienced programmer to develop a full-scale, hand-coded program to solve a selected application. They will then compare the development time and utility of the hand-coded program with that of a user-developed program.

To demonstrate reusability, Marques and his colleagues need to show that new control mechanisms are not needed for each new application. (This is critical because they admit that the cost of building mechanisms and their associated knowledge acquisition tools is too large if they need a special tool for each new application.) Each of the nine programs that they developed used between two and six mechanisms; thirteen mechanisms were used altogether. Seven applications used the dialog manager, six used the select mechanism, and five used the classify mechanism.

Marques and his colleagues report that "[o]ne of [their] biggest problems is getting people to ’make contact’ with Spark’s activity model. People buried in the details of ’real work’ have difficulty understanding generic, abstract models of their tasks unless they helped to create the models." [MARQ92, p. 29] In fact, the example given in their report, an example of sifting through the hierarchy of problem/solution models, shows a bewildering array of possibilities. More discouraging is that, upon selecting an incorrect mechanism, the user can be

led through a tedious, repetitious cycle of programming by example only to be sent back to the beginning to select a more appropriate mechanism. The basic approach appears to be programming by educated guess, followed by trial and error refinement.

Marques and his colleagues have developed the most sophisticated, computer-assisted software development tools reported in the literature to date. The tools compose and refine an application from a set of reusable components. The composition method employs knowledge encoded within the tools, coupled with knowledge elicited from a domain expert. The reusable components and the elicitation, generation, and run-time tools define an architecture into which elicited knowledge can be encoded. Instead of relying on standards to define an open architecture, the developers have constructed a closed environment.

The system produced by Marques and his colleagues meet the criteria for an automatic programming system, as defined by Barstow in 1985, with one exception. The reliability of programs produced by the DEC system cannot be assessed because a given application program is never really completed. The program continues to be refined, growing smarter, and presummably more reliable, with use.

The discussion of prototyping approaches presented in this section illustrates that all prototyping involves reuse, sometimes of components, sometimes of knowledge encoded in a knowledge base, a transformation program, or a simulator, but most often of a combination of components and knowledge. Successful operational and evolutionary prototyping approaches rely mainly on composition of large, reusable components, coupled with knowledge elicited from human programmers or users. The form of reusable components varies from open systems composed of standard software functions that are accessible via

standard interfaces to closed systems of components that are integrated into an expert system, that are accessible via heuristic classification strategies, and that can be modified through an interactive dialog between a user and an expert system. In the case of open prototyping systems, methods for eliciting and representing composition rules are not well understood nor widely available. In the case of knowledge-based prototyping systems, methods for eliciting and representing knowledge, for selecting a possible solution, and for modifying the selected solution to meet specific application details are all integrated into the prototyping system. Of course, the cost of building such closed prototype systems is quite high -- all the reusable solutions, elicitation software, and analysis and classification algorithms must be constructed before prototypes can be built. Even when not intended to evolve into operational software systems, prototyping systems can provide an architecture, or context, into which reusable components and knowledge can be fitted and evaluated.

VI. Conclusions

Reuse in the software industry, as with any engineering discipline, holds the key to increased productivity among practitioners and to improved quality among software products. Although the concept of software reuse was identified as early as 1969, the progress achieved within the software industry, while significant from some points of view, disappoints most reuse advocates. Some may argue that revised economic incentives and solid management commitment will enhance software reuse practice, but this paper has described a long list of hard, technical barriers that impede software reuse. In fact, the paper has reported a historical perspective that shows software reuse rates peaking at about 50% in 1984, and then dwindling to about 33% by 1989. Worse, the components being reused in 1989 were both smaller and simpler than those reused in 1984. What can account for this trend?

Much of the early progress in reuse relied upon a well understood application domain (business information systems) and a specific, well defined programming architecture (COBOL programs on mainframe computers). Advances in technology, such as new programming languages (Ada and C), new programming paradigms (object oriented programming and graphical user interfaces), and fast, cheap, desktop computers, have washed away progress made in reuse among COBOL business applications. Reuse principles could not be generalized and applied easily in new environments. To move software reuse back to and beyond the 50% peak realized in 1984, new approaches, drawing on the successful early experiences, are needed.

The present paper has identified three keys to successful software reuse: 1) a well understood application domain, 2) large-grained reusable components, and 3) a definite system

model or architecture into which reusable components can be fitted. The system architecture should be general enough so that reusable components are not necessarily limited to those written in a specific, programming language.

The first requirement for successful software reuse, a well understood domain, can be approached through a domain analysis and modeling process that incorporates knowledge-based tools and techniques. This paper proposed integrating domain analysis into the requirements engineering process. Further, the paper showed how such integration could be achieved by proposing a specific mapping between a domain analysis process described by Prieto-Diaz and a requirements engineering process defined in a seminar on requirements engineering held at George Mason University in the fall of 1992. The paper went on to identify several tools and techniques that could be used to aid the integrated domain analysis and requirements engineering processes. These tools and techniques mainly supported the elicitation, acquisition, and representation of knowledge.

The second requirement for productive software reuse, large-grained reusable components, is the subject of current research. Several object oriented paradigms are moving toward the concept of frameworks or ensembles to represent a collection of related classes that support a large, reusable concept. In fact, collections of classes are used to implement some of the key reusable components in the industry today; these are the so called "megaobjects" such as X-Windows, the Motif graphical user interface, TCP/IP communications software, and SQL interface libraries. The trend toward larger, reusable components is also evident in automated programming systems, such as the user programming system developed by researchers at the Digital Equipment Corporation. In DEC’s approach, reusable control mechanisms are constructed and then inserted into a library where searches, conducted by a user, for a control structure to

support specific application requirements are assisted by an expert system. Once a control structure is selected, a related elicitation program acquires the necessary knowledge to solve the specific application problem and then represents that knowledge in a form that can be used by the pre-defined control structure. During use, the application program can be refined with additional knowledge, even to the point of selecting an alternate control structure.

These two approaches, one called an open architecture and the other a closed expert system, both address reuse at a large grain size. Each approach has advantages and disadvantages, as outlined in the paper, that go to the heart of the relationship between prototyping and knowledge acquisition and representation. The open architecture can more easily accept components developed by different approaches and using different languages; however, the transformation between domain knowledge is made by a human programmer. The closed expert system can use pre-defined knowledge, guided by an interaction with a domain expert, to produce a solution to a specific application problem; however, the pre-defined knowledge must already be encoded into the system, and encoded in a form that meets the requirements of the expert system. This pre-encoding of knowledge is expensive, and the number of sources that can encode the knowledge is probably rather small.

This dilemma, open versus closed architecture, describes the problem faced by the industry when the third prerequisite for successful software reuse, a well defined system architecture, is considered. There can be no doubt that a definite architecture is necessary for reuse: consider the success of UNIX, MSDOS, and, maybe, MS Windows. These are each a specific architecture that enables reusable software; but such software cannot be easily moved from one of these architectures to the others. Will the industry have winners and losers? Will

one architecture predominant and others fade? Such is our history.

This paper has advocated that some form of prototyping be used to evaluate reusable components, and also that prototyping can be a source of previously evaluated reusable components; however, the paper does not come down on the side of a specific architecture, or even on the side of a closed or open architecture. Should an open architecture be used, the problem of choosing particular standards must be addressed. (This is a problem worthy of separate consideration in another study.) Open architectures also provide little help with some reuse problems such as classification, location, and retrieval. Closed, expert system architectures can help solve such problems. Of course, closed architectures may suffer from the reuse population problem.

More research is required concerning the problem of bridging the conceptual gap between domain knowledge representations and prototype representations. In fact, the forms of representing knowledge (logic programming, production rules, frames, and semantic networks) are rather limited no matter to what use the knowledge will be put. For example, although expert systems can elicit knowledge from domain experts and then encode that knowledge into a knowledge base suitable to support reasoning, no means presently exists to translate textual requirements statements, gathered from users by analysts, into a form suitable for storing in a knowledge base or for comparing against a knowledge base. Until further progress is made on these issues, knowledge-based systems can at best provide a limited form of automated assistance to human analysts.

VII. References and Bibliography

The following citations were used in the paper as references or as general background material. To aid understanding, the citations are divided into four categories: 1) General Software and Requirements Engineering, 2) Reuse, 3) Domain Knowledge Acquisition, Representation, and Analysis, and 4) Prototyping and Automatic Programming. Within each category the citations are arranged alphabetically, using the citation code as a key. Each citation code, enclosed in square brackets [], is composed of the first four characters of the first author's last name (except where the author's last name is shorter than four letters) and the last two digits of the year in which the reference was published. Where the citation encodes to more than one identical value a single lowercase alphabetic character is used to distinguish the synonyms.

General Software and Requirements Engineering

[BOEH87] B. Boehm, "Improving Software Productivity", *COMPUTER*, September 1987, pp. 43-75.

[BROO87] F. Brooks, "No Silver Bullet, Essence and Accidents of Software Engineering", *COMPUTER*, April 1987, pp. 10-19.

[CURT88] B. Curits, et al., "A Field Study of the Software Design Process for Large Systems", *Communications of the ACM*, November 1988, pp. 1268-1286.

[DAVI90] A. Davis, *Software Requirements Analysis and Specification*, Prentice Hall, 1990, 516 pages.

[FREE80] P. Freeman, "Requirements Analysis and Specification: The First Step", *Advances in Computer Technology*, American Society of Mechanical Engineers, 1980, pp. 79-85.

[PALM92] J. Palmer and N. Fields, "An Integrated Environment for Requirements Engineering", *IEEE Software*, May 1992, pp. 80-85.

[RAMA86] C. Ramamoorthy, et al., "Programming in the Large", *IEEE Transactions on Software Engineering*, July 1986, pp. 769-783.

[RAME92] B. Ramesh and V. Dhar, "Supporting Systems Development by Capturing Deliberations During Requirements Engineering", *IEEE Transactions on Software Engineering*, June 1992, pp. 498-510.

[RZEP85] W. Rzepka and Y. Ohno, "Requirments Engineering Environment: Software Tools for Modeling User Needs", *COMPUTER*, April 1985, pp. 9-12.

[SAGE90] A. Sage and J. Palmer, *Software Systems Engineering*, John Wiley and Sons, 1990, 511 pages.

[YADA88] S. Yadar, et al., "Comparison of Analysis Techniques for Information Requirements Determination", *Communicatons of the ACM*, September 1988, pp. 1090-1097.

Reuse

[BIGG87] T. Biggerstaff and C. Richter, "Reusability Framework, Assessment, and Directions", *IEEE Software*, March 1987, pp. 41-49.

[BIGG89] T. Biggerstaff, "Design Recovery for Maintenance and Reuse", *COMPUTER*, July 1989, pp. 36-49.

[BURT87] B. Burton, et al., "The Reusable Software Library", *IEEE Software*, July 1987, pp. 25-32.

[CALD91] G. Caldiera and V. Basili, "Identifying and Qualifying Reusable Software Components", *COMPUTER*, February 1991, pp. 61-69.

[CAVA89] M. Cavaliere, "Reusable Code at the Hartford Insurance Group", in *Software Reusability Volume II Applications and Experience*, ACM Press, 1989, pp. 131-141.

[CURT89] B. Curtis, "Cognitive Issues in Reusing Software", in *Software Reusability Volume II Applications and Experience*, ACM Press, 1989, pp. 269-287.

[COX92] B. Cox, The Economics of Software Reuse, a lecture given in INFT 821 at George Mason University on Oct. 13, 1992.

[DISA91] Defense Information Systems Agency, *Defense Software Repsitory System General Information*, available from the DoD Center for Software Reuse Operations, 500 N. Washington St., Suite 101, Falls Church, VA 22046.

[GRIS91] M. Griss, et al., "The Economics of Software Reuse", *OOPSLA '91 Conference Proceedings*, October 1991, pp. 264-270.

[HORO84] E. Horowitz and J. Munson, "An Expansive View of Reusable Software", *IEEE Transactions on Software Engineering*, September 1984, pp. 477-487.

[ISCO88] N. Iscoe, "Domain-Specific Reuse: An Object-Oriented and Knowledge-Based Approach", an updated version of an article in the *Proceedings of the Workshop on Software Reuse* held in October 1987, pp. 299-308.

[JACO91] I. Jacobson and F. Lindstrom, "Re-engineering of Old Systems to an Object Oriented Architecture, *OOPSLA '91 Conference Proceedings*, October 1991, pp. 340-350.

[JONE84] T. Jones, "Reusability in Programming: A Survey of the State of the Art", *IEEE Transactions on Software Engineering*, September 1984, pp. 488-493.

[KERN84] B. Kernighan, "The Unix System and Software Reusability", *IEEE Transactions on Software Engineering*, September 1984, pp. 513-518.

[LANG84] R. Langergan and C. Grasso, "Software Engineering with Reusable Designs and Code", *IEEE Transactions on Software Engineering*, September 1984, pp. 498-501.

[LEDB85] L. Ledbetter and B. Cox, "Software-ICs", *BYTE*, June 1985, pp. 28-36.

[LENZ87] M. Lenz, et al., "Software Reuse Through Building Blocks", *IEEE Software*, July 1987, pp. 34-42.

[LEWI91] J. Lewis, et al., "An Empirical Study of the Object Oriented Paradigm and Software Reuse", *OOPSLA '91 Conference Proceedings*, October 1991, pp. 264-270.

[MATS84] Y. Matsumoto, "Some Experiences in Promoting Reusable Software Presentation in Higher Abstraction Levels", *IEEE Transactions on Software Engineering*, September 1984, pp. 502-512.

[MEYE87] B. Meyer, "Reusability: The Case for Object Oriented Design", *IEEE Software*, March 1987, pp. 50-64.

[NOVA92] G. Novak, et al., "Negotiated Interfaces for Software Reuse", *IEEE Transactions on Software Engineering*, July 1992, pp. 646-652.

[PRIE87] R. Pietro-Diaz and P. Freeman, "Classifying Software for Reusability", *IEEE Software*, January 1987, pp. 6-16.

[RICE89] J. Rice and H. Schwetman, "Interface Issues in a Software Parts Technology",in *Software Reusability Volume I Concepts and Models*, ACM Press, 1989, pp. 125-139.

[SELB89] R. Selby, "Quantitative Studies of Software Reuse", in *Software Reusability Volume II Applications and Experience*, ACM Press, 1989, pp. 213-233.

[STAN84] T. Standish, "An Essay on Software Reuse", *IEEE Transactions on Software Engineering*, September 1984, pp. 494-497.

[VOLP89] D. Volpano and R. Kieburtz, "The Templates Approach to Software Reuse", in *Software Reusability Volume I Concepts and Models*, ACM Press, 1989, pp. 247-255.

[WIRF90] R. Wirfs-Brock and R. Johnson, "Surveying Current Research in Object-Oriented Design", *Communications of the ACM*, September 1990, pp. 104-102.

[WOOD87] S. Woodfield, et al., "Can Programmers Reuse Software?", *IEEE Software*, July 1987, pp. 52-59.

Domain Knowledge Acquisiton, Representation, and Analysis

[ARAN89] G. Arango, "Domain Analysis - From Art Form To Engineering Discipline", *ACM*, 1989, pp. 152-159.

[BOOS86] J. Boose, "ETS: A System for the Transfer of Human Expertise", in *Knowledge Based Problem Solving*, Prentice Hall, 1986, pp. 68-111.

[BROO83] R. Brooks, "Towards a Theory of the Comprehension of Computer Programs", *International Journal of Man-Machine Studies*, 1983, pp. 543-554.

[BUCH78] B. Buchanan and E. Feigenbaum, "DENDRAL and META-DENDRAL: Their Applications Dimension", Artificial Intelligence, November 1978.

[CHOR90] D. Chorafas, *Knowledge Engineering*, Van Nostrand Reinhold, 1990, 380 pages.

[CLAN86] W. Clancey, "Heuristic Classification", in *Knowledge Based Problem Solving*, Prentice Hall, 1986, pp. 1-67.

[COOK87] N. Cooke and J. McDonald, "The Application of Psychological Scaling Techniques to Knowledge Elicitation for Knowledge-based Systems", *International Journal of Man-Machine Studies*, vol. 26, 1987, pp. 530-550.

[DAVI82] R. Davis and D. Lenat, *Knowledge-Based Systems in Artificial Intelligence*, McGraw Hill, 1982.

[EEPE92] Ee-Peng L. and V. Cherkassky, "Semantic Networks and Associative Databases", *IEEE Expert*, August 1992, pp. 31-40.

[FICK87] S. Fickas, "Automating the Requirements Analysis Process: An Example", paper presented at some *IEEE Conference* held in 1987, pp. 58-67.

[FICK92] S. Fickas and R. Helm, "Knowledge Representation and Reasoning in the Design of Composite Systems", *IEEE Transactions on Software Engineering*, June 1992, pp. 470-482.

[FIKE85] R. Fikes and T. Kehler, "The Role of Frame-Based Representation in Reasoning", *Communications of the ACM*, September 1985, pp. 904-920.

[GENE85] M. Genesereth and M. Ginsberg, "Logic Programming", *Communications of the ACM*, September 1985, pp. 933-941.

[HASS83] N. Hass and G. Hendrix, "Learning by Being Told: Acquiring Knowledge for Information Management", in *Machine Learning: An Artificial Intelligence Approach*, Tioga Press, 1983.

[HAYE85] F. Hayes-Roth, "Rule-Based Systems", *Communications of the ACM*, September 1985, pp. 921-932.

[KELL55] G. Kelly, *The Psychology of Personal Constructs*, Norton, 1955.

[KITT87] C. Kitto and J. Boose, "Choosing Knowledge Acquisition Strategies for Application Tasks", in *Proceedings*

Western Conference On Expert Systems, IEEE Computer Society, June 1987, pp. 96-103.

[KONT88] J. Kontos and J. Cavouras, "Knowledge Acquisition from Technical Texts Using Attribute Grammars", *The Computer Journal*, June 1988, pp. 525-530.

[MART90] J. Martin and W. Tsai, "N-Fold Inspection: A Requirements Analysis Technique", *Communications of the ACM*, February 1990, pp. 225-232.

[MICH80] R. Michalski , "Pattern Recognition as Rule-guided Inductive Inference", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, no. 4, 1980.

[MUSE89] M. Musen, *Automated Generation of Model-Based Knowledge Acquisition Tools*, Morgan Kaufman Publishers, 1989, 293 pages.

[NAGY92] G. Nagy, et al., "A Prototype Document Image Analysis System for Technical Journals", *COMPUTER*, July 1992, pp. 10-24.

[OBAN87] R. O'Bannon, "An Intelligent Aid to Assist Knowledge Engineers with Interviewing Experts", in *Proceedings Western Conference On Expert Systems*, IEEE Computer Society, June 1987, pp. 31-43.

[PRIE87a] R. Prieto-Diaz, "Domain Analysis For Reusability", *IEEE*, 1987, pp. 23-29.

[RICH92] C. Rich and Y. Feldman, "Seven Layers of Knowledge Representation and Reasoning in Support of Software Development", *IEEE Transactions on Software Engineering*, June 1992, pp. 451-469.

[RINE91] D. Rine, "A Formal Approach to Software Domain Modeling of Requirements Specification Using a Basis as Primitive Domain Types", unpublished manuscript, October 1991, 28 pages.

[SHEM87] I. Shemer, "Systems Analysis: A Systemic Analysis of a Conceptual Model", *Communications of the ACM*, June 1987, pp. 506-512.

[ANDR92] S. Andriole, *Rapid Application Prototyping - The Storyboard Approach to User Requirements Analysis*, QED Technical Publishing Group, 1992, 336 pages.

[BALZ83] R. Balzer, et al., "Software Technology in the 1990's: Using a New Paradigm", *COMPUTER*, November 1983, pp. 39-45.

[BARS85] D. Barstow, "Domain-Specific Automatic Programming", *IEEE Transactions on Software Engineering*, November 1985, pp. 1321-1336.

[DAVI92] A. Davis, "Operational Prototyping: A New Development Approach", *IEEE Software*, September 1992, pp. 70-78.

[GARL92] D. Garlan, et al., "Using Tool Abstraction to Compose Systems", *COMPUTER*, June 1992, pp. 30-38.

[HOLT91] R. Holt, et al., *Object Oriented Computing: Looking Ahead to the Year 2000*, Information Technology Research Center, TR-9101, University of Toronto, April 1991, 123 pages.

[KAIS89] G. Kaiser and D. Garlan, "Synthesizing Programming Environments From Reusable Features", in *Software Reusability Volume II Applications and Experience*, ACM Press, 1989, pp. 35-55.

[LEE91] S. Lee and S. Sluizer, "An Executable Language for Modeling Simple Behavior", *IEEE Transactions on Software Engineering*, June 1991, pp. 527-543.

[LEWI89] T. Lewis, et al., "Prototypes from Standard User Interface Management Systems", *COMPUTER*, May 1989, pp. 51-60.

[LOWR92] M. Lowry, "Software Engineering in the Twenty-First Century", *AI Magazine*, Fall 1992, pp. 71-87.

[LUQI89] Luqi, "Software Evolution Through Rapid Prototyping", *COMPUTER*, May 1989, pp. 13-25.

[MARQ92] D. Marques, et al., "Easy Programming", *IEEE Expert*, June 1992, pp. 16-29.

[NEIG89] J. Neighbors, "DRACO: A Method for Engineering Reusable Software Systems", in *Software Reusability Volume I Concepts and Models*, ACM Press, 1989, pp. 295-319.

[RICH88] C. Rich and R. Waters, "The Programmer's Apprentice: A Research Overiview", *COMPUTER*, November 1988, pp. 10-25.

[RICH88a] C. Rich and R. Waters, "Automatic Programming: Myths and Prospects", *COMPUTER*, August 1988, pp. 40-51.

[TANI89] M. Tanik and R. Yeh, "Rapid Prototyping in Software Developement", *COMPUTER*, May 1989, pp. 9-13.

[ZAVE84] P. Zave, "The Operational Versus the Conventional Approach to Software Development," *Communications of the ACM*, February 1984, pp. 104-118.